NAUTILUS: Fishing for Deep Bugs with Grammars

Cornelius Aschermann
Ruhr-Universität Bochum
cornelius.aschermann@rub.de

Tommaso Frassetto
Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

Patrick Jauernig
Technische Universität Darmstadt
patrick.jauernig@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

Daniel Teuchert
Ruhr-Universität Bochum
daniel.teuchert@rub.de

***Abstract*—Fuzz testing is a well-known method for efficiently identifying bugs in programs. Unfortunately, when programs that require highly-structured inputs such as interpreters are fuzzed, many fuzzing methods struggle to pass the syntax checks: interpreters often process inputs in multiple stages, first syntactic and then semantic correctness is checked. Only if both checks are passed, the interpreted code gets executed. This prevents fuzzers from executing "deeper" — and hence potentially more interesting — code. Typically, two valid inputs that lead to the execution of different features in the target program require too many mutations for simple mutation-based fuzzers to discover: making small changes like bit flips usually only leads to the execution of error paths in the parsing engine. So-called *grammar fuzzers* are able to pass the syntax checks by using Context-Free Grammars. Feedback can significantly increase the efficiency of fuzzing engines and is commonly used in state-of-the-art mutational fuzzers which do not use grammars. Yet, current grammar fuzzers do not make use of code coverage, i.e., they do not know whether any input triggers new functionality.**

In this paper, we propose NAUTILUS, a method to efficiently fuzz programs that require highly-structured inputs by combining the use of grammars with the use of code coverage feedback. This allows us to recombine aspects of interesting inputs, and to increase the probability that any generated input will be syntactically and semantically correct. We implemented a proof-of-concept fuzzer that we tested on multiple targets, including ChakraCore (the JavaScript engine of Microsoft Edge), PHP, mruby, and Lua. NAUTILUS identified multiple bugs in all of the targets: Seven in mruby, three in PHP, two in ChakraCore, and one in Lua. Reporting these bugs was awarded with a sum of 2600 USD and 6 CVEs were assigned. Our experiments show that combining context-free grammars and feedback-driven fuzzing significantly outperforms state-of-the-art approaches like AFL by an order of magnitude and grammar fuzzers by more than a factor of two when measuring code coverage.

I. INTRODUCTION

Software controls more and more aspects of the modern life. Hence, the importance of software testing is increasing at a similar pace. Human-written tests (e.g., unit tests) are an important part of the software development life cycle; yet, many software projects have no or limited testing suites due to a variety of reasons. Even for projects with comprehensive testing suites, tests usually revolve around *expected inputs* in order to test the *intended functionality* of code. However, *unexpected inputs* are one of the primary attack vectors used to exploit applications using their *intended functionality*, whereas automated software testing excels at finding inputs with *unexpected characteristics* that can be leveraged to trigger vulnerabilities.

One popular approach to automatically test programs is *fuzzing*, i.e., automatically testing programs by generating inputs and feeding them to the program while monitoring crashes and other unexpected conditions. In recent years, many different fuzzers were developed, covering a variety of approaches and goals. General-purpose fuzzers [18] usually rely on low-level binary transformations to generate new inputs, thus, they struggle with programs which only accept highly structured files, such as interpreters for scripting languages. Binary transformations generate inputs that struggle to pass initial lexical and syntactic analysis [36] and reach the code that executes after those checks, i.e., the *deep code*.

An intuitive solution to this problem is to use (context-free) grammars to generate syntactically-correct inputs. Previous works [5], [17], [36], [47] use this approach, but they do not leverage instrumentation feedback, which allows the fuzzer to distinguish inputs that reach a new part of the code base from inputs that reach no new code.

Leveraging feedback led to a great improvement in the performance of general-purpose fuzzers. One of the most popular feedback-oriented fuzzers is AFL [19], which was used to identify bugs in hundreds of applications and tools. Using code coverage feedback, AFL is able to intelligently combine interesting inputs to explore deeper code, which would take an unreasonable amount of time without feedback. In contrast, AFL struggles with heavily-structured file formats since it is optimized for binary formats and does not support grammars. Note that AFL can be provided with a list of strings, which it will try to use to generate inputs. However, this list does not support any kind of grammar-like semantics.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23412
www.ndss-symposium.org

Most coverage-driven fuzzers, including AFL, require a *corpus* of inputs which they use as a basis to start the fuzzing process. A good-quality corpus is crucial to the performance and effectiveness of the fuzzer: any code path that is used by the corpus does not have to be discovered by the fuzzer and can be combined with other inputs from the beginning. Getting such a high-quality corpus is not trivial: if the language accepted by the target application is widely used, one approach is to crawl publicly available examples from the Web [49] or from public code repositories. However, those examples are likely to skew towards very commonly-used parts of the grammar, which use well-tested parts of the target application. Naturally, security researchers often want to test features which are rarely used or were introduced very recently, which are more likely to lead to bugs in the target application. Acquiring a corpus to test those features is clearly harder, while writing examples manually is very expensive.

Goals and contributions. In this paper, we present the design and implementation of a fuzzing method that combines description-based input generation and feedback-based fuzzing to find bugs *deep* in the applications' semantics, i.e., bugs that happen after lexical and syntactical checks. Our prototype implementation of this concept called NAUTILUS requires no corpus, only the source code of an application and a grammar to generate inputs for it. One can start fuzzing with NAUTILUS using publicly-available grammars [6]. The fuzzing process can then be fine-tuned by removing uninteresting parts of the grammar and adding additional information about the language, e.g., by incorporating function names and parameter types taken from the language documentation, which can be easily automated. Additionally, NAUTILUS allows the user to extended grammars with additional script. These scripts allow NAUTILUS to generate any decidable input language to further improve its ability to generate semantically correct inputs.

However, NAUTILUS does not just generate interesting initial inputs. The fuzzing process itself also leverages the grammar by performing high-level semantic transformations on the inputs, e.g., swapping an expression for a different expression in a program. By combining these mutations with the coverage feedback, NAUTILUS can create a corpus of semantically interesting and diverse inputs and recombine them in a way that drastically increases the probability of finding new inputs which are both syntactically and semantically valid. As we evaluate in Section VI, those two advantages give NAUTILUS a significant advantage over state-of-the-art fuzzers. Additionally, NAUTILUS was able to find new bugs in all the targets it was tested on: seven in mruby[1], three in PHP, one in Lua, and two in ChakraCore.

In summary, our contributions in this paper are:

- We introduce and evaluate NAUTILUS, the first fuzzer that combines grammar-based input generation with feedback-directed fuzzing. NAUTILUS significantly improves the efficiency and effectiveness of fuzzing on targets that require highly-structured inputs—without requiring any corpus. To increase expressiveness, NAUTILUS supports Turing-complete scripts as an extension to the grammar for input language descriptions. This can be used to create descriptions for complex, non-context-free input languages.
- We describe and evaluate several grammar-based mutation, minimization and generation techniques. By combining coverage feedback and grammar-based splicing, NAUTILUS is able to generate syntactically and often semantically correct programs, outperforming traditional purely-generational fuzzers that spend significant time generating and testing semantically invalid inputs.
- We found and reported several security bugs in multiple widely-used software projects which no other fuzzer in our evaluation found.

To foster research on this topic, we release our fuzzer at https://github.com/RUB-SysSec/nautilus.

[1]CVE-2018-10191, CVE-2018-10199, CVE-2018-11743, CVE-2018-12247, CVE-2018-12248, and CVE-2018-12249.

II. BACKGROUND

A. Fuzzing

Fuzzing is a quick and cost-effective technique to find coding flaws in applications. Traditionally, there are two ways for fuzzers to generate input for target applications: mutation or generation.

For mutational fuzzing [15], [19], a well-formed corpus of *seed* inputs, e.g., test cases, is modified using genetics-inspired operations like bit flips, or recombination of two inputs (*splicing*). These modifications can be either random (brute force), or guided using a heuristic. More advanced techniques are either taint-based [24], [27], [29], [43], [50], symbolic [26], or concolic [31], [32], [46] (a portmanteau of concrete and symbolic). Taint-based fuzzers try to track down input bytes that, e.g., influence function arguments, while symbolic analysis treats some of the input bytes symbolically, and uses symbolic execution to explore new paths. Concolic execution combines these techniques: dynamic analysis (e.g., guided or taint-based fuzzers) is used to get as many new paths as possible, then, corresponding concrete values are passed to a symbolic execution engine to take new branches (guarded by more complex checks) to explore new paths. These new paths are the input for the next iteration of dynamic analysis. A popular mutation-based fuzzer is the heuristically-guided fuzzer AFL [18]. AFL uses new basic block transitions as a heuristic for coverage.

In contrast, generation-based fuzzers can generate input based on a given specification, usually provided as a model or a grammar. For example, if the target application is an interpreter, the underlying grammar of the programming language can be used to generate syntactically valid input. This allows them to pass complex input processing, while semantic checks remain challenging for these approaches. Furthermore, many generation-based fuzzers not only require a grammar, but also a corpus [36], [47]. Creating such a corpus may be cumbersome, since the corpus should ideally contain valid as well as invalid test cases, since together they can be recombined to valid, crashing inputs [36].

Orthogonally, fuzzers generally can be divided into black-box and white-box fuzzers. While black-box fuzzers do not require insight in an application, only needing a (large) corpus, white-box fuzzers leverage extensive instrumentation and analysis techniques to overcome conditional branches and track

code paths taken. White-box approaches try to systematically explore new code paths that are harder to find for black-box fuzzers, however, increasing analysis also induces a decreasing number of test cycles per second.

B. Context-Free Grammars

Applications often require highly-structured input, which a conventional mutation-based fuzzer cannot easily provide. Context-free grammars (CFGs) are well-suited to specify highly structured input languages. Here we give a short definition of CFGs and an introduction to how they can be used to describe input languages. Intuitively, a CFG is a set of production rules of the form "Some variable X (non-terminal symbol) can be replaced by the following array of strings (terminal-symbols) and variables". Additionally, a special start non-terminal specifies where to begin applying these rules. The input language described by the CFG is the set of all strings that can be derived by applying any number of rules until no more non-terminals are present.

More formally, a CFG is defined as a tuple: $G = (N, T, R, S)$ with:

- N is a finite set of non-terminals. Non-terminals can be thought of as intermediate states of the language specification.
- T is a finite set of terminal symbols. N and T are disjoint.
- R is a finite set containing the *production rules* of the form $A \rightarrow a$ where $A \in N$ and $a \in^* T \cup N$
- $S \in N$ is a non-terminal which is the start symbol. Every word generated by the CFG needs to be derivable from S.

Since the left-hand side of each rule consists of *exactly* one non-terminal, the possible derivations only depend on one non-terminal and no context, therefore these grammars are called *context free*.

To derive a string, a matching production rule, i.e., one with the respective non-terminal on the left-hand side, has to be applied to the start symbol S. As long as the right-hand side of that rule contains a non-terminal, another derivation step is executed. For each step, one non-terminal is replaced by the right-hand side of a rule matching the non-terminal.

Example II.1 shows a possible input generation given a grammar G1.

Example II.1. Consider the following grammar (G1):

```
N: {PROG, STMNT, EXPR, VAR, NUMBER}
T: {a, 1, 2, =, return 1}
R: {
```

$$
\begin{aligned}
\textit{PROG} &\rightarrow \textit{STMT} && (1)\\
\textit{PROG} &\rightarrow \textit{STMT}\ ;\ \textit{PROG} && (2)\\
\textit{STMT} &\rightarrow \text{return 1} && (3)\\
\textit{STMT} &\rightarrow \textit{VAR} = \textit{EXPR} && (4)\\
\textit{VAR} &\rightarrow \text{a} && (5)\\
\textit{EXPR} &\rightarrow \textit{NUMBER} && (6)\\
\textit{EXPR} &\rightarrow \textit{EXPR} + \textit{EXPR} && (7)\\
\textit{NUMBER} &\rightarrow 1 && (8)\\
\textit{NUMBER} &\rightarrow 2 && (9)
\end{aligned}
$$

```
}
S: PROG
```

Therefore, one possible derivation from G1 would be: `PROG` $\xrightarrow{(1)}$ `STMT` $\xrightarrow{(4)}$ `VAR = EXPR` $\xrightarrow{(5)}$ `a = EXPR` $\xrightarrow{(6)}$ `a = NUMBER` $\xrightarrow{(8)}$ `a = 1`. Numbers over arrows denote applied production rules. The derived string is `a=1`.

Each string generated by a CFG can be represented by its *derivation tree*. A derivation tree t of a CFG G is a tree whose vertices are labeled by either non-terminals or terminals. The root of t is labeled with the start symbol, all terminal vertices are labeled with terminals from G [52]. NAUTILUS mostly operates on these derivation trees instead of the trees' string representation which we call *unparsed* strings. Derivation trees are NAUTILUS's internal representation for inputs to which it applies structural mutations. However, as many common language constructs are not context free (e.g., checksums, or generating proper XML as the opening and the closing tags need to contain the same identifier), we extend upon CFGs by allowing additional scripts to be used to transform the input.

Since the set of production rules must contain all (relevant) non-terminals and terminals, in the following we define CFGs only through their production rules and a start symbol. To distinguish between non-terminals and terminals, we use uppercase names for non-terminals.

III. CHALLENGES

Designing a fuzzer requires thorough consideration in order to minimize the effort required from the user and maximize effectiveness of the fuzzer. In particular, we identified four key aspects that are desirable:

C1: Generation of syntactically and semantically valid inputs. Generated inputs need to pass the syntactic and semantic checks of the target application to reach the next stages of computation. The subset of syntactically and semantically valid inputs is usually much smaller than the set of all possible inputs [49]. Therefore, it is often hard for fuzzers to go "deeper" and find bugs in the application logic that is guarded by the input validation. Additionally, in many cases the input language cannot be modeled by simpler formalisms such as CFGs.

C2: Independence from corpora. Current fuzzers often need an initial corpus of inputs, i.e., a set of seed files. Even with well-known software, where a corpus is usually available, acquiring a corpus which targets the new and obscure parts of the application is hard. Acquiring a corpus for internal

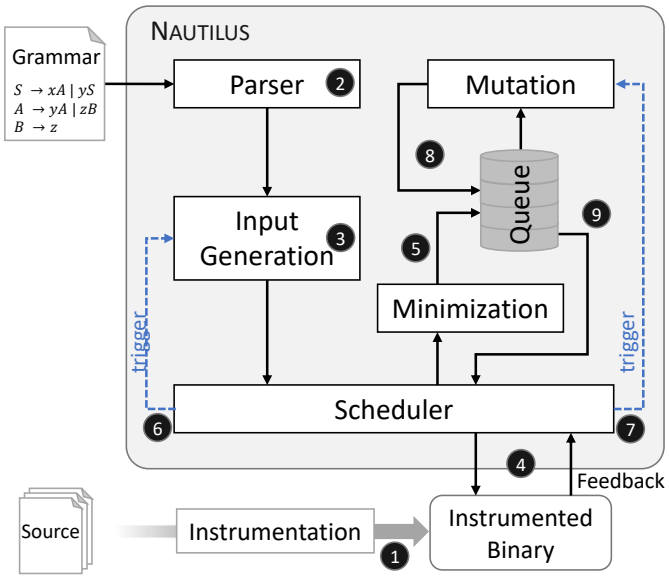


Fig. 1: High-level overview of NAUTILUS.

or unreleased software is even harder. Hence, presence of a (proper) corpus should not be required to fuzz the application. However, many software projects have a specification of allowed inputs (or a grammar) that can be leveraged instead. In addition, techniques that try to combine seed corpora with input specifications typically need to restrict their input languages to enable parsing, hence, reducing the usefulness of seed corpora when using input specifications.

C3: High coverage of target functionality. Achieving a high coverage in the target application is desirable to find a higher number of bugs. This mainly relates to two aspects: (1) passing input processing, and (2) steering analysis to explore new paths.

C4: Good performance. Fast testing cycles, which imply a high number of tested inputs per fuzzing window, is one of the key aspects to increase effectiveness of fuzzers. To ensure high execution rates, the inputs need to be small and the generation method needs to be fast.

We designed NAUTILUS with these challenges in mind: grammar-only input derivation tackles C1 and C2, while we use feedback-driven input generation to address C3, and steering input derivation length as well as minimization of interesting inputs to take on C4. In the following section, we explain the design of these high-level concepts.

IV. DESIGN OF NAUTILUS

A high-level overview of the approach is shown in Figure 1. The first step required to use NAUTILUS is to compile the source code of the target application using our instrumentation ❶ to give feedback on coverage information while running. Then, the fuzzer process itself is started, parses the grammar the user provided ❷, then generates a small amount (1000) of random initial inputs from scratch ❸ and passes them to the scheduler. Then, NAUTILUS tests whether that newly generated input triggers any new coverage by executing the instrumented binary ❹. If it did, NAUTILUS minimizes it using the grammar, and adds it to the queue ❺. Based on whether new paths can still be explored, the scheduler will either trigger mutation of existing inputs ❻ or derivation of a new input ❼. For inputs in the queue, mutations based on the grammar are applied. The mutation methods include techniques like replacing subtrees with newly generated ones or combining trees that found different interesting features of the target application. After mutation, the input is added to the queue ❽ and used in subsequent analysis runs ❾. This architecture allows us to combine the strength of both grammar fuzzing and feedback fuzzing to recombine existing interesting inputs into more semantically interesting inputs. In the following, we explain the process of generation, minimization, and mutation process in more detail.

A. Generation

The generation algorithm should produce inputs which use distinct aspects of the grammar in order to maximize coverage (Challenge C3). Our fuzzer internally only uses the tree representation instead of the unparsed string representation of the word. This allows us to operate on the tree and to define custom unparsing routines. We use the grammar rules to derive random trees from the set derivation trees. If there are multiple rules for one non-terminal, there are multiple ways to select a rule for further derivation. In this paper we evaluate two different approaches: *naive generation* and *uniform generation*.

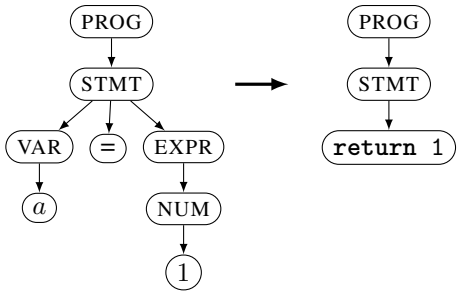
The naive generation approach is to randomly pick one of the applicable rules. As shown below in Example IV.1, this can lead to a lot of similar inputs being generated. For this reason, we augmented this generation approach with a filter that checks whether the generated input was already generated in the recent past.

Example IV.1 (Naive Generation)**.** When generating a tree from grammar G1 from Example II.1, there are two rules for the `STMT` non-terminal. If rule 3 is picked, the generation immediately terminates. If rule 4 is chosen, the generation continues, and multiple different trees can be generated. If both rules are picked with 50% probability, half of the generated inputs are `return 1`.

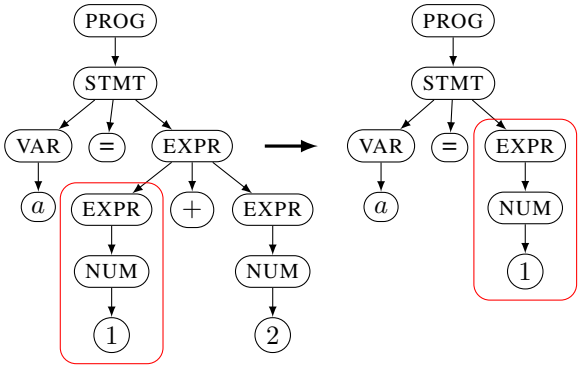
Our other approach, *uniform generation*, is able to uniformly draw trees from the set of all possible trees as proposed by McKenzie [39]. This approach avoids the strong bias introduced by the structure of the grammar. This algorithm takes a CFG and an integer $n \geq 0$ and returns strings of length n, derived uniformly from the given input grammar. This algorithm picks the rules based on the number of distinct subtrees that they can generate: for each non-terminal n, for each possible length l, for each production rule r of n the amount of possible subtrees $p(n, l, r)$ is calculated. As an example, a rule that can generate four different subtrees is picked twice as often as a rule which can generate two subtrees.

B. Minimization

After an interesting input was found, NAUTILUS tries to produce a smaller input that triggers the same new coverage. Minimized inputs have the advantage of a shorter execution time and a smaller set of potential mutations during further processing. We use two different approaches to minimize inputs that found new paths:



Example IV.2 (Subtree Minimization)**.** The Subtree Minimization is executed on `a = 1`, replacing the subtree of `STMT` with a smaller one.



Example IV.3. Recursive Minimization is executed on `a = 1 + 2`, which contains a recursive `EXPR`: both the whole right-hand side, as well as the individual numbers, are derivable from `EXPR`. Using either of the two numbers instead of the addition yields a valid minimized tree.

Subtree Minimization aims to make subtrees as short as possible while still triggering new path transitions. For each nonterminal, we generate the smallest possible subtree. Then, we sequentially replace each node's subtree with the smallest possible subtree at this position, and check if the new transitions are still triggered by the changed input. If the transitions are still taken, the changed input replaces the original one, otherwise the changed input is discarded (see Example IV.2).

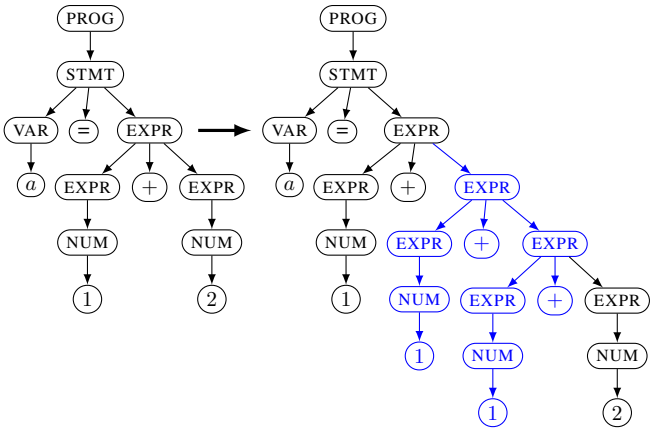
Recursive Minimization This strategy is applied *after* Subtree Minimization. Its goal is to reduce the amount of recursions by identifying recursions and replacing them one at a time. Example IV.3 displays how a nested expression is minimized.

C. Mutation

After an input was minimized, NAUTILUS uses multiple mutation methods to generate new tests. Unless specified otherwise, whenever we pick some element randomly, we pick uniformly amongst all options.

Random Mutation picks a random node of a tree and replaces it with a randomly-generated new subtree rooted in the same nonterminal. The size is chosen randomly and the maximum size of the subtree is a configuration parameter.

Rules Mutation sequentially replaces each node of the input tree with one subtree generated by all other possible rules. This mutation resembles the deterministic phase used by AFL.



Example IV.4. This tree contains a recursion (an `EXPR` node has `EXPR` child nodes). Random Recursive Mutation randomly repeats this subtree recursion (two times in the example) and inserts the result in the already existing recursion. This turns the simple `a = 1 + 2` into the more complex `a = 1 + (1 + (1 + (1 + 2)))`.

Random Recursive Mutation picks a random recursion of a tree and repeats that recursion 2^n times ($1 \leq n \leq 15$). This creates trees with higher degree of nesting. An example application of this mutation can be seen in Example IV.4.

Splicing Mutation combines inputs that found different paths by taking a subtree from one interesting input and placing it in another input: it replaces one of the subtrees with a "fitting" subtree from another tree in the queue. To do so, it picks a random internal node, which becomes the root of the subtree to be replaced. Then it picks from a tree in the queue a random subtree that is rooted in the same nonterminal to replace the old subtree.

AFL Mutation performs mutations that are also used by AFL such as bit flips or interesting integers. The AFL Mutation operates on strings, so subtrees are converted into text form before this mutation is applied. This mutation can produce invalid trees which are sometimes interesting to discover parser bugs. This mutation consists of several different sub-mutations:

Bit Flips flip single or multiple bits at once;

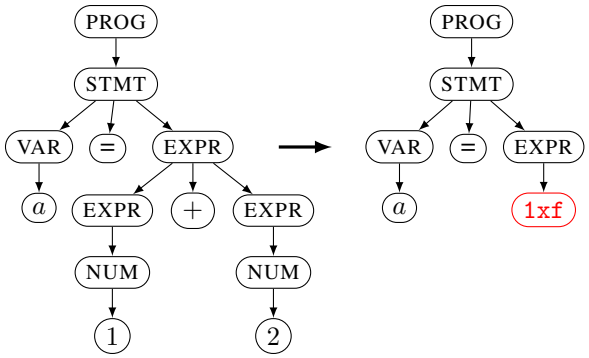
Arithmetic Mutations add or subtract interesting values to numeric values in the string;

Interesting Values replaces parts of the string with interesting values.

Afterwards, the mutated terminal string is stored as a new *Custom Rule* with the same originating non-terminal and added to the tree, replacing the original subtree. Custom Rules are not added to the grammar, they are saved locally with the tree. This process is depicted in Example IV.5.

D. Unparsing

After NAUTILUS obtained a candidate derivation tree, NAUTILUS needs to generate an actual input from it. This step of turning the derivation tree into a binary file is typically



Example IV.5. In this example, AFL Mutation alters the terminal string of the subtree of the topmost `EXPR` by flipping some bits from `1 + 2` to `1xf` (not valid according to the grammar). Then, a custom rule `EXPR` $\rightarrow$ `"1xf"` is created which replaces the subtree, resulting in the input `a = 1xf`;

called unparsing. For true CFGs, this process is straightforward. The unparsing function is recursively defined to concatenate all unparsed subtrees. However, many real-world input grammars are not context free, as mentioned earlier. Hence, we extend rule definitions by an additional unparsing script that can perform arbitrary computation on the results of all unparsed subtrees. Scripting support is one of the big strengths of our generative approach, as there is no need to parse inputs. Therefore, we can freely venture beyond decidable grammars without any disadvantages. Approaches such as Skyfire [49] or IFuzzer [47] are restricted to grammars which support parsing. Example IV.6 shows how this technique can produce syntactically and semantically valid XML documents.

Example IV.6. Assume that the grammar for XML contains a rule that specifies a simplified tag. Each tag has an ID and a BODY. The corresponding rule is: $TAG \rightarrow$ <ID>$BODY$</ID> In CFGs the opening and the closing ID are independent. Therefore, we might produce inputs such as `"<a>foo</b>"` by performing the concatenation on the children: `["<","a",">","foo","</","b",">"]`. By extending CFGs with unparsing scripts, the rule turns into $TAG \rightarrow ID$,$BODY$ with the unparsing function `lambda |id,body| "<"+id+">"+body+"</"+id+">"`. Thus, we are able to reuse the ID twice to produce a valid XML tag.

V. Implementation

NAUTILUS is implemented in Rust and its overall architecture is similar to AFL. We use the mruby interpreter to execute the scripts embeddable in our extended grammars. Similar to AFL, NAUTILUS requires the target program to be instrumented. It fuzzes the target in a number of phases. The following sections describe those processes in more detail.

A. Target Application Instrumentation

NAUTILUS shares the concept of AFL's source-code instrumentation: a 64 KB bitmap is shared with the application. A custom compiler pass adds Instrumentations which updates this bitmap based on information about basic blocks transitions performed in the target application. Additionally, the compiler pass adds some code that runs the application in a forkserver to increase the rate at which inputs can be executed.

B. ANTLR Parser

NAUTILUS accepts grammar inputs as either JavaScript Object Notation (JSON), the natural grammar representation used by NAUTILUS, or grammars written for ANother Tool for Language Recognition (ANTLR) [1], since more than 200 ANTLR grammars for a number of programming languages are already publicly available [6]. In order to support ANTLR grammars, we integrate an *ANTLR Parser* component that converts the ANTLR grammar into the native NAUTILUS format. The parser can convert most grammars automatically. In some cases, ANTLR grammars do not specify whitespace, since it is not relevant during parsing; however, whitespace is relevant during input generation. Thus, one typically has to add spaces in a few key rules in those grammars.

C. Preparation Phase

NAUTILUS precomputes some data based on the grammar provided before the fuzzing begins. This data includes:

$\min(n)$ for each non-terminal n, the minimum number of rules that need to be applied to generate a string which uses n as the start non-terminal. This data is used by the Rules Mutation (Section IV-C).

$p(n, l, r)$ For each non-terminal n, for each possible length l, for each production rule r of n, the number of possible subtrees rooted in n, using r as the first rule, applying l rules. This information is needed for the Uniform Generation (Section IV-A).

$p(n, l)$ For each non-terminal n, for each possible length l, the number of possible subtrees. This represents how many possible subtrees can be generated when applying l rules starting from the non-terminal n, or, in other words, the number of derivation trees with root n and l edges. This information is also needed for the Uniform Generation (Section IV-A).

The algorithm for calculating the minimum length for non-terminals is very similar to the one proposed by Purdom [42]. The other values are calculated using the algorithm proposed by McKenzie [39].

D. Fuzzing Phase

Figure 2 shows the workflow of NAUTILUS during the fuzzing phase. After generating some initial inputs, the *Scheduler* decides which input should be tried next: either (i) mutate an existing input with a certain mutation, or (ii) generate a new input from scratch. The scheduler controls a queue which contains all generated or mutated inputs that are still considered interesting, i.e., each of them triggers at least one transition between basic blocks in the application that no other input triggers.

The scheduler processes every item in the queue sequentially. Each item in the queue has a `state` which indicates how it will be processed when taken from the queue. The state can be one of these values:

`init` If an input triggered a new transition, it is saved in the queue with the `init` state. When the scheduler selects an item in the `init` state, the item is minimized

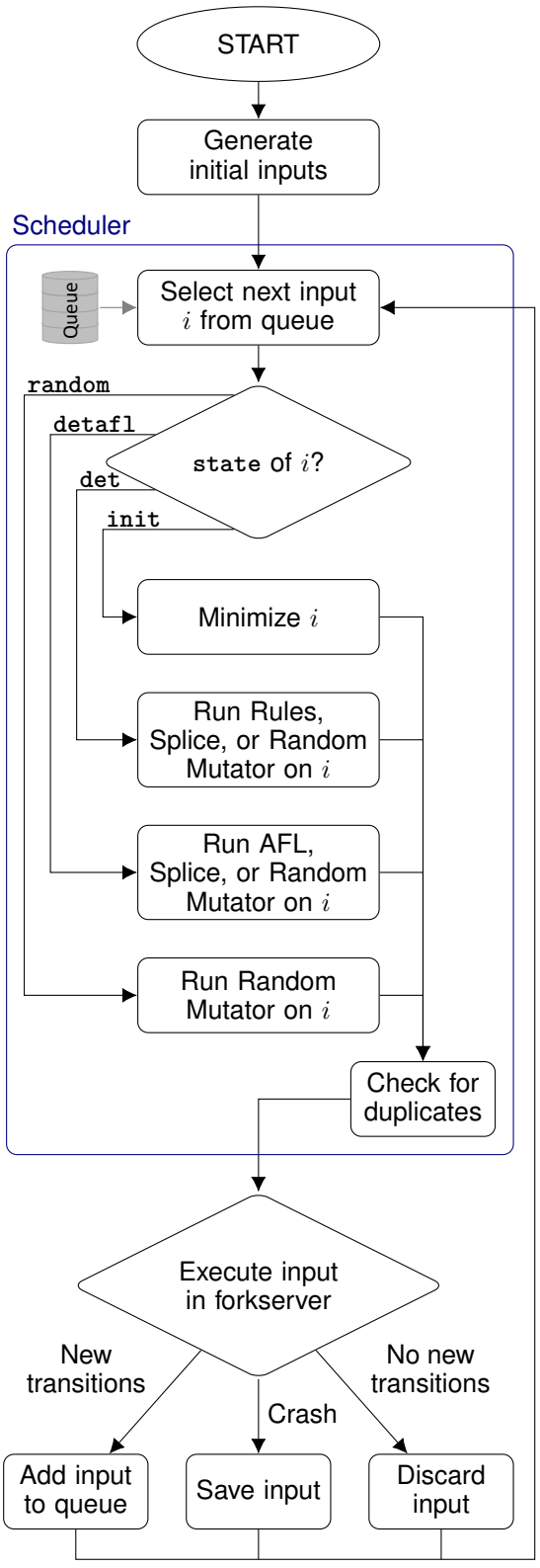


Fig. 2: Workflow of the fuzzing phase.

(Section IV-B). After finishing the minimization of an item, its state is set to `det`.

`det` Items in the `det` are mutated using the Rules Mutator, the Random (Recursive) Mutator, and the Splice Mutator (see Section IV-C). When the Rules Mutator is done with an item because no more mutations of that type are possible, the item moves to the `detafl` state.

`detafl` Items in the `det` are mutated using the AFL Mutator, the Random (Recursive) Mutator, and the Splice Mutator. When the AFL Mutator is done with an item, it moves to the `random` state.

`random` This is the final state of each entry. Only the *Random Mutation*, *Random Recursive Mutation*, and *Splice Mutation* are applied on this entry. In contrast to AFL, we do not finish each stage before we continue with the next input. Instead NAUTILUS only spends a short amount of time (typically a few seconds) on each input, before we continue with the next one. Therefore, we quickly explore those inputs that are very likely to yield new coverage while not spending too much time on unproductive inputs. This allows us to achieve an effect similar to AFLFast [25].

After an input tree is selected for execution, it is unparsed to an input string. Then, the target program is run with this input using a fork server similar to the one used by AFL, which can start the target application in a highly-efficient way. There are three possible states that can follow: (i) the target application crashes during the execution, then, the binary representation of the input that caused the crash is saved in a separate folder, (ii) the input caused the target application to take a new path, then, the tree representation of the input is added to the queue, or (iii) the input did not trigger any new transition and the input is discarded.

VI. EVALUATION

We tested NAUTILUS on four real-world applications and it found vulnerabilities in all of them. We chose four programming language interpreters as targets since these had well documented grammars. The efficiency of our prototype was also evaluated against other state of the art fuzzers.

Our evaluation aimed at answering the following research questions:

RQ 1 Can NAUTILUS identify new bugs in real-life applications?

RQ 2 Is NAUTILUS more efficient than other state-of-the-art fuzzers?

RQ 3 How much does the use of grammars improve the fuzzing efficiency for target applications with highly structured inputs?

RQ 4 When using grammars, how much does the use of feedback increase the fuzzing performance?

RQ 5 Does our complex generation method, which requires more computational power than a naive approach, actually increase fuzzing performance, and if so, how much?

RQ 6 How much does each of the mutation methods used contribute to find new paths?

Section VI-B describes the bugs NAUTILUS found and discusses RQ1. Section VI-C evaluates the efficiency of NAU-

Target	Type	CVE
mruby	Use after free caused by integer overflow	CVE-2018-10191
	Use after free in `initialize_copy`	CVE-2018-10199
	Use of uninitialized pointer in `hash.c`	CVE-2018-11743
	Segmentation fault in `mrb_class_real`	CVE-2018-12247
	Segmentation fault in `cfree`	CVE-2018-12248
	Heap buffer overflow caused by `Fiber::transfer`	CVE-2018-12249
	Stack overflow (not fixed yet)	none yet
PHP	Division by Zero triggered by range() caused by a type conversion.	-
	Segmentation fault in `zend_mm_alloc_small`	-
	Stack overflow caused by using too many parameters in a function call.	-
ChakraCore	Wrong number of arguments emitted in JIT-compiled code	-
	Segmentation fault in out-of-memory conditions	-
Lua	Type confusion	-

TABLE I: Vulnerabilities found by NAUTILUS in our targets

TILUS and discusses RQ2, RQ3, and RQ4. Section VI-D evaluates our generation method and discusses RQ5, while Section VI-E evaluates our mutation methods and discusses RQ6.

A. Experimental Setup

For our evaluation we chose four widely-used scripting languages: Ruby, Lua, PHP, and JavaScript:

- For Ruby, we chose the mruby implementation [8] since it is used by Shopify in their infrastructure and they have an open bug bounty program (see Section VI-B for details). We fuzzed various versions of mruby during the first half of 2018. The performance experiments were performed using the version from Git commit `14c2179` on the official mruby repository [9].
- For Lua, we used version 5.3.4 from the official site [13].
- For PHP, we used version 7.2.6 from the official distribution network [11].
- For JavaScript, we chose the ChakraCore implementation [2], since it was made public more recently. We used the code from Git commit `cf87c70` for performance measurements and the code from commit `550d2ea` for our fuzzing campaign.

For our performance evaluation we used 14 identical machines, each with an Intel Core i5-650 CPU clocked at 3.2 GHz, 4 GB of RAM, and Ubuntu 16.04.4 LTS. Each fuzzer was only allowed to use one core; we only ran one fuzzer on any machine at any given time to avoid interferences. Each test was performed 20 times (12 times for IFuzzer [47]) to enable a statistical analysis of the results.

We based our grammars on existing ANTLR grammars [6]. We performed a set of changes to improve the performance: as mentioned earlier, some cases required adding whitespaces, as they are typically discarded during tokenization and not part of the grammar. Additionally, we replaced the rules to generate identifiers with a list of strings retrieved from the documentation or the program itself. Lastly, we often significantly shrunk the grammar for strings and number literals; otherwise, the fuzzer would spend a lot of time exploring random literals that add very little interesting information.

B. Vulnerabilities Identified

To answer RQ1 we evaluated our prototype by fuzzing our four target applications. Our fuzzer was able to find new bugs in all four, while none of the other fuzzers did during the evaluation period. All bugs were reported and acknowledged by the various vendors. The vulnerabilities are summarized in Table I and described below.

mruby: Mruby is an interpreter for the Ruby programming language with the intention of being lightweight and easily embeddable [8]. In total, we found 7 bugs in mruby, including two use-after-free vulnerabilities[2], two segmentation faults, one use of an uninitialized pointer, one heap buffer overflow, and a stack overflow (see Table I). 6 CVEs were assigned so far. Reporting these bugs was awarded with a sum of 2,600 USD from the shopify-scripts bug bounty program [16].

Case Study: *Finding CVEs*. *Given the bug bounty program and the ease of the reporting process, we performed a more thorough analysis of mruby. We started by inspecting previous security issues and we noticed that nearly all bugs did not rely on special strings or non-trivial integers. Therefore, we built a grammar that only contains a small set of identifiers, integers and strings. We also significantly reduced the variance in the language, e.g., by including only one of the multiple ways to invoke methods. Using this grammar allowed us to find multiple CVEs.*

PHP: PHP is a popular general-purpose scripting language that is especially suited to web development [11]. NAUTILUS found three bugs in PHP: a division by zero, a segmentation fault, and a stack overflow (see Table I). The bugs were not considered security bugs by the PHP developers since they require "obviously malicious" code. For this reason, no CVEs where obtained for the three identified bugs. However, those bugs could be triggered in a sandboxed PHP environment and all lead to a crash.

[2] In a use-after-free vulnerability, a program calls `free` on a pointer, then dereferences the pointer and uses the memory again. An adversary can force the application to allocate some other data at that address and then run the faulty code on data of the adversary's choosing.

Target	Baseline Coverage	Fuzzer	Mean	Median	**Median New Coverage Found**	Std Deviation	Skewness	Kurtosis
ChakraCore	14.7%	NAUTILUS	34.0%	34.1%	**19.4 pp**	0.60 pp	−0.29	−0.44
		NAUTILUS - No Feedback	18.6%	18.5%	**3.8 pp**	0.24 pp	1.42	0.53
		AFL	15.8%	15.8%	**1.1 pp**	0.27 pp	0.10	−0.58
		IFuzzer	15.9%	16.0%	**1.3 pp**	0.20 pp	−1.08	0.35
mruby	25.7%	NAUTILUS	53.7%	53.8%	**28.1 pp**	1.60 pp	−0.16	−0.38
		NAUTILUS - No Feedback	37.7%	37.8%	**12.1 pp**	0.34 pp	−0.81	−1.01
		AFL	28.0%	27.6%	**1.9 pp**	1.28 pp	2.36	4.20
PHP	2.2%	NAUTILUS	11.7%	12.3%	**10.0 pp**	2.17 pp	−0.65	−0.43
		NAUTILUS - No Feedback	6.1%	6.1%	**3.9 pp**	0.09 pp	0.08	−1.69
		AFL	2.2%	2.2%	**0.0 pp**	0.00 pp	−1.40	0.61
Lua	39.4%	NAUTILUS	66.7%	66.6%	**27.2 pp**	1.33 pp	−0.11	−0.72
		NAUTILUS - No Feedback	47.9%	47.8%	**8.4 pp**	1.02 pp	0.11	−1.80
		AFL	54.4%	54.6%	**15.2 pp**	0.54 pp	−1.80	2.42

TABLE II: Statistics about branch coverage. The new coverage found is the additional coverage that was found by the fuzzer w.r.t. the initial corpus. "pp" stands for "percentage points". Note: AFL was able to find some coverage on PHP, but the results round to zero.

ChakraCore: ChakraCore is the JavaScript engine used by the web browser Edge [2]. Our fuzzer identified two bugs in ChakraCore: one bug in the Just In Time (JIT) compiler where the wrong number of arguments were emitted for a function call, and a segmentation fault caused by out of memory conditions. The bug only affected the Linux branch of ChakraCore, and therefore was not eligible for a bug bounty.

Lua: Lua is a lightweight, embeddable scripting language [13]. NAUTILUS identified a bug caused by a type confusion, which causes a crash in Lua. The issue was reported to the Lua mailing list. Our example input relies on a debug feature, therefore the bug was not considered a security issue.

C. Evaluation Against Other State-of-the-art Fuzzers

To answer RQ2, we ran our tool NAUTILUS and other state-of-the-art fuzzers on the four targets mentioned in Section VI-B over multiple runs with identical durations. To measure how much of an application's code was tested by the fuzzer we use *branch coverage*, i.e., the percentage of branches of the applications that were executed at least once during the fuzzing run. A fuzzer which achieves a high code coverage can often also identify more bugs, since it executes more possibly faulty code.

In order to evaluate our approach, we compared to two state-of-the-art fuzzers using different approaches, namely, AFL [19] for feedback-directed fuzzing and IFuzzer [47] for grammar-based fuzzing. Since IFuzzer is not as flexible as AFL and NAUTILUS and only supports JavaScript, we only tested it on ChakraCore. We provided AFL with a dictionary containing the same strings we used for the NAUTILUS grammars. Moreover, for each target, we generated 1000 inputs from the grammar, and provided them as a seed corpus. We chose to run AFL with the generated inputs after we verified this corpus lets AFL discover significantly more code than 10 hand-picked examples containing real-world code. We also provided the same corpus to IFuzzer. We used the *naive generation* mode on NAUTILUS, AFL version 2.52b, and IFuzzer from commit `8debd78`. We ran each configuration of fuzzer and target 20 times for 24 hours each.

To avoid relying on the different internal reporting methods of the fuzzers, we configured them to save all interesting inputs as files[3] and we measured the branch coverage using standard code coverage tools, namely GCOV [4] and Clang's Source-based Code Coverage [3].

The results of our experiments are summarized in Table II and displayed in Figure 3. The baseline denotes the coverage that was found by our generated corpus itself with no fuzzer interaction. NAUTILUS is able to find significantly higher amounts of additional coverage: while AFL and IFuzzer find between 0 and 1.9 percentage points of additional coverage on ChakraCore, mruby and PHP, NAUTILUS discovers between 10.0 and 28.1 percentage points of additional coverage. For Lua, AFL discovers 15.2 additional percentage points, while NAUTILUS discovers 27.2 additional percentage points. As we provide AFL with a good dictionary and an extensive corpus of cases to learn from, we consider this setup as a very strong baseline and a significant bar to meet. This is also evident by the fact that, even though IFuzzer is based on a good grammar, it barely exceeds the performance of AFL.

We performed a Mann-Whitney U test as recommended by Arcui et al. [21] to ensure statistical significance and we report the results in Table III. In all cases, our worst run was better than the best run of all other tools. Due to this, the p-Values obtained by the U test are extremely small, and we can exclude the possibility that the observed differences are the result of random chance. All statistics were computed using the Python scipy [38] and numpy [40] libraries.

To address the relative merit of grammar-based input generation (RQ3) and feedback-directed fuzzing (RQ4) we disabled the coverage feedback mechanism in NAUTILUS and we ran it in the same environment as the full version. This experiment is meant to prove that the combination of feedback and grammar fuzzing does indeed create a significant performance advantage, everything else being equal (performance of the implementation and grammars). We consider this experiment as a proxy to evaluate fuzzers such as Peach [10] or Sulley

[3]Since IFuzzer does not support saving input cases, we modified it slightly to add this functionality.

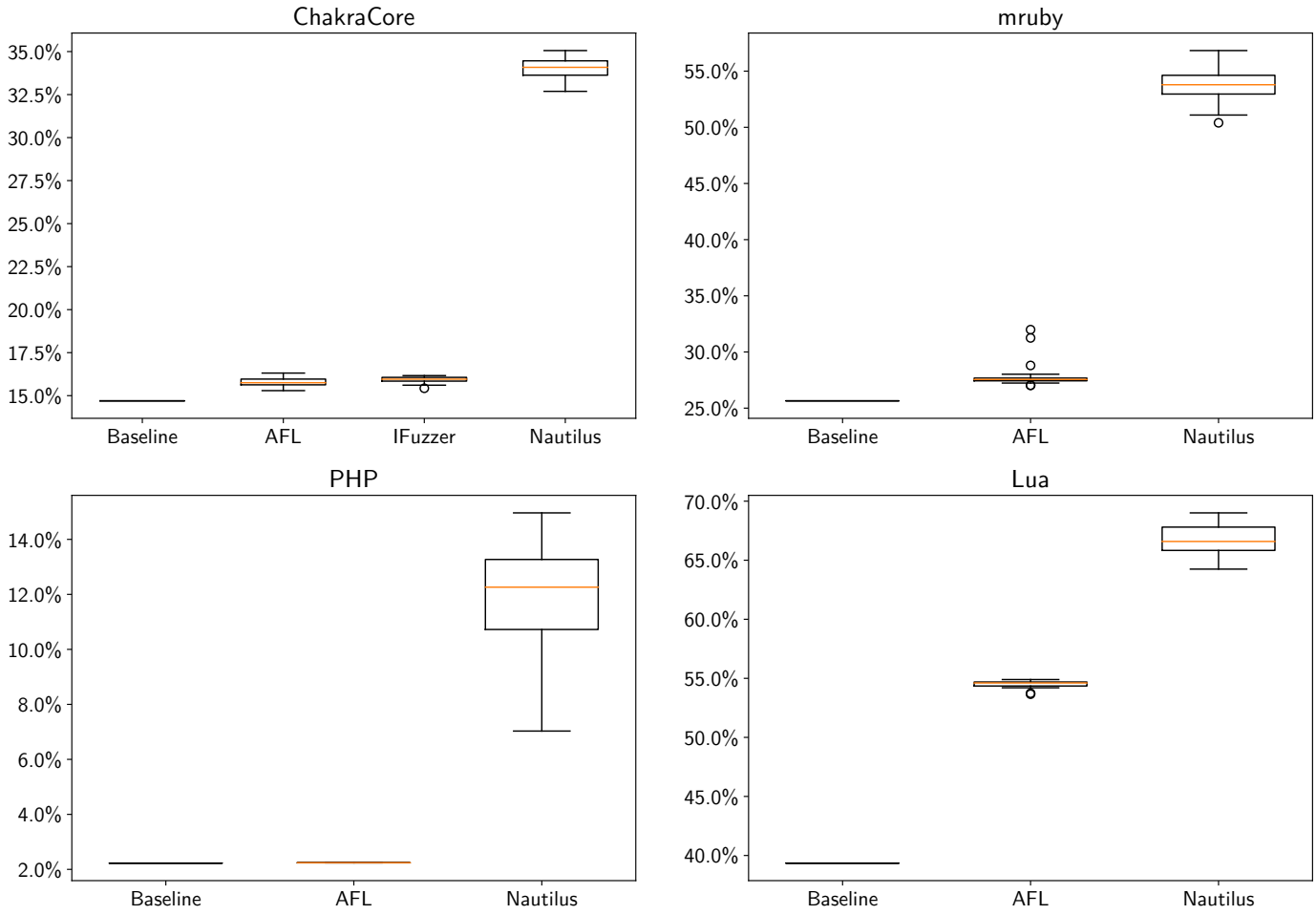


Fig. 3: Branch coverage generated by our corpus of 1000 generated inputs *(Baseline)* and by the different fuzzers over 20 runs of 24 hours each.

Experiment	Effect size ($\Delta = \bar{A} - \bar{B}$)	p-Value ($\times 10^{-6}$)
ChakraCore (vs. AFL)	18.3 pp	0.033
ChakraCore (vs. IFuzzer)	18.1 pp	1.6
mruby (vs. AFL)	26.2 pp	0.033
PHP (vs. AFL)	10.0 pp	0.017
Lua (vs. AFL)	12.0 pp	0.033

TABLE III: Confirmatory data analysis of our experiments. The effect size is the difference of the medians in percentage points. Due to storage requirements, we only performed 12 runs for IFuzzer, hence the slightly higher p-Value. In the case of PHP, AFL generated the exact same coverage multiple times, which explains the slight change in the p-Value compared to the other configurations. In all cases the effect size is relevant and the changes are highly significant: the p-Value is about ten thousand times smaller than the usual bound of $p < 0.05$.

[17]. We could not directly evaluate these tools since they need manually written generators instead of grammars.

The results can be seen in Figure 4 as well as Table II (labeled *No feedback*). In all cases, we observed that using a purely generational grammar fuzzer resulted in significant coverage increases over AFL.

As expected, blind grammar fuzzing improves upon the mutational fuzzing performed by AFL (In the case of mruby, by more than one order of magnitude), even when a large number of seeds is given. This shows that grammar fuzzing remains highly relevant even after AFL and related tools have been published (RQ3). Yet, adding feedback to the grammar fuzzing approach results in even greater improvements: In all cases we found more than twice as many new branches than the blind grammar fuzzer. Therefore, we conclude that adding feedback to grammar fuzzing is a very important step to improve the performance of fuzzing (RQ4).

Case Study: *Feedback Grammar Fuzzing*. *When fuzzing mruby,* NAUTILUS *automatically learned the following interesting code fragment:* `ObjectSpace.each{|a| a.method(...) }` *It allows the fuzzer to test a method call on any object in existence. This greatly amplifies the chance of finding the right receiver for a method. Any time the fuzzer guesses a correct method name, this construct immediately produces new coverage. Then the fuzzer can incrementally learn how to construct valid arguments to this specific call. Lastly, if the fuzzer needs a specific receiver object, it only has to create the object somewhere in the input, as any object will receive the method call. An AFL-style fuzzer is not able to make use of this. It is not able to construct*

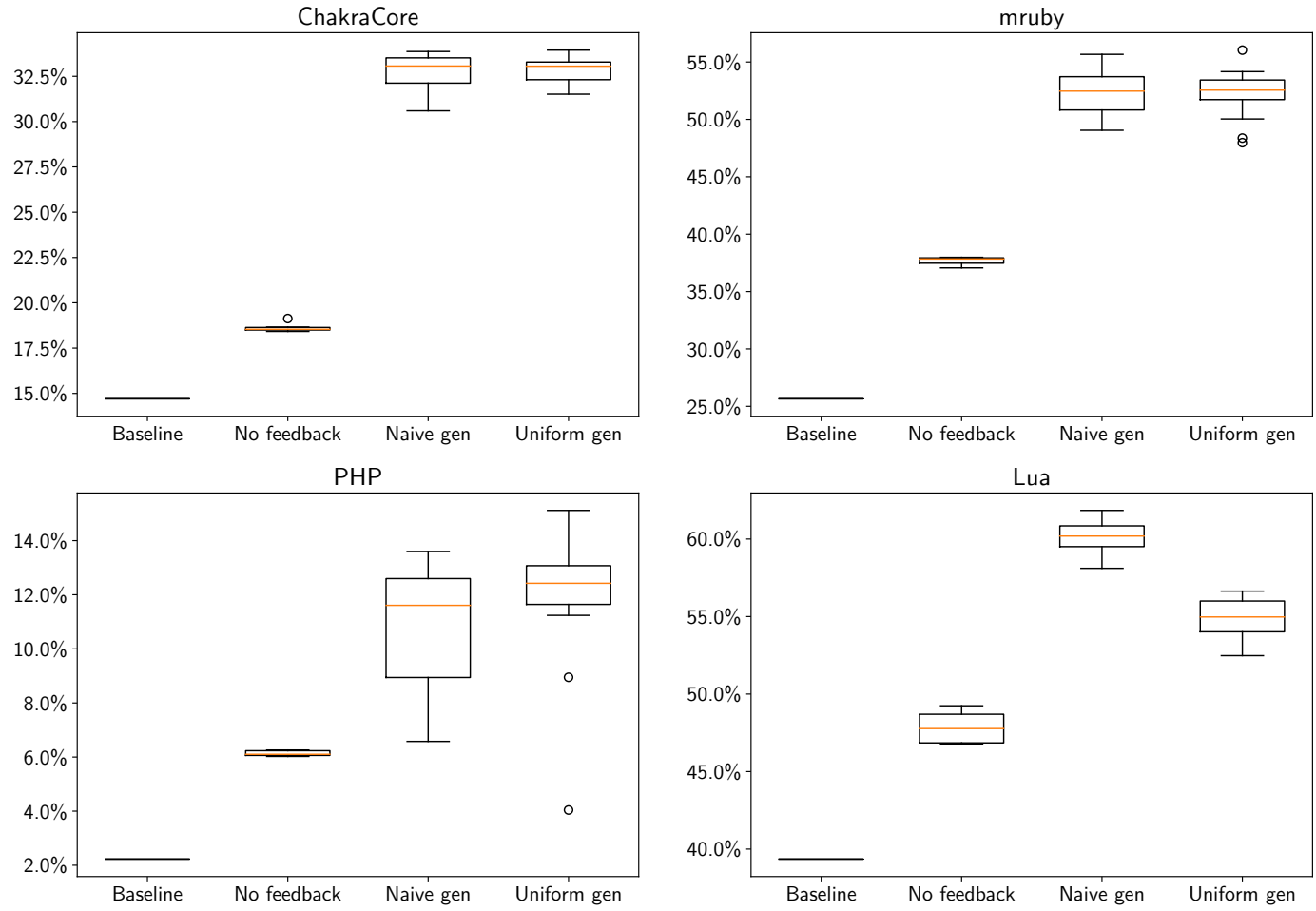


Fig. 4: Branch coverage generated by our corpus *(Baseline)* and by different configurations of NAUTILUS.

complex arguments for the call, even if a set of valid method names is given as a dictionary. A blind grammar fuzzer is unable to reliably produce this gadget with an interesting call inside. NAUTILUS *used this gadget to find two CVE's that would have been exceedingly hard to find using either blind grammar fuzzing or AFL-style fuzzing without a proper grammar.*

D. Evaluation of Generation Methods

To analyze our generation methods and answer RQ5, we analyzed the performance of NAUTILUS using naive and uniform generation (see Section IV-A), using the same configuration as in Section VI-C. Figure 4 shows the difference between the naive generation and the uniform generation methods.

The results are very similar. The naive generation achieves very similar results for ChakraCore and mruby; it performs slightly better on PHP, while it performs significantly worse on Lua. This proves that the naive generation method, when combined with the simple duplicate filter, performs very similarly to the more complex uniform generation, making the additional complexity of the latter unnecessary.

E. Evaluation of Mutation Methods

To answer RQ6 and to analyze the efficacy of the different mutation methods, our fuzzer keeps a counter for each mutation method, the minimization methods, and the generation. These counters are increased if the corresponding method created an input that found a new code path. Note that counters will only be increased by one, regardless of the amount of new transitions discovered by any specific input. Using the same configuration of Section VI-C (using coverage feedback) our fuzzer was run on each of the four targets. Figure 6 shows the values of those counters at the end of each run.

Additionally, we evaluated the usefulness of the various methods by computing the relative contribution of each method over the 24 hours of each run. Due to the diminishing number of new paths after the initial part of each run, we collected the data in differently-sized bins: 1-minute bins for the first 16 minutes, 2-minute bins until 30 minutes from the beginning, 3-minute bins until 1 hour from the beginning, 5-minute bins until the 3 hour mark, 10-minute bins until the 6 hour mark, then 20-minute bins until the end. The result is shown in Figure 5. It can be seen that splicing becomes more and more relevant over time, as the basic mutation methods slowly fail to produce more semantically valid inputs. Eventually splicing of

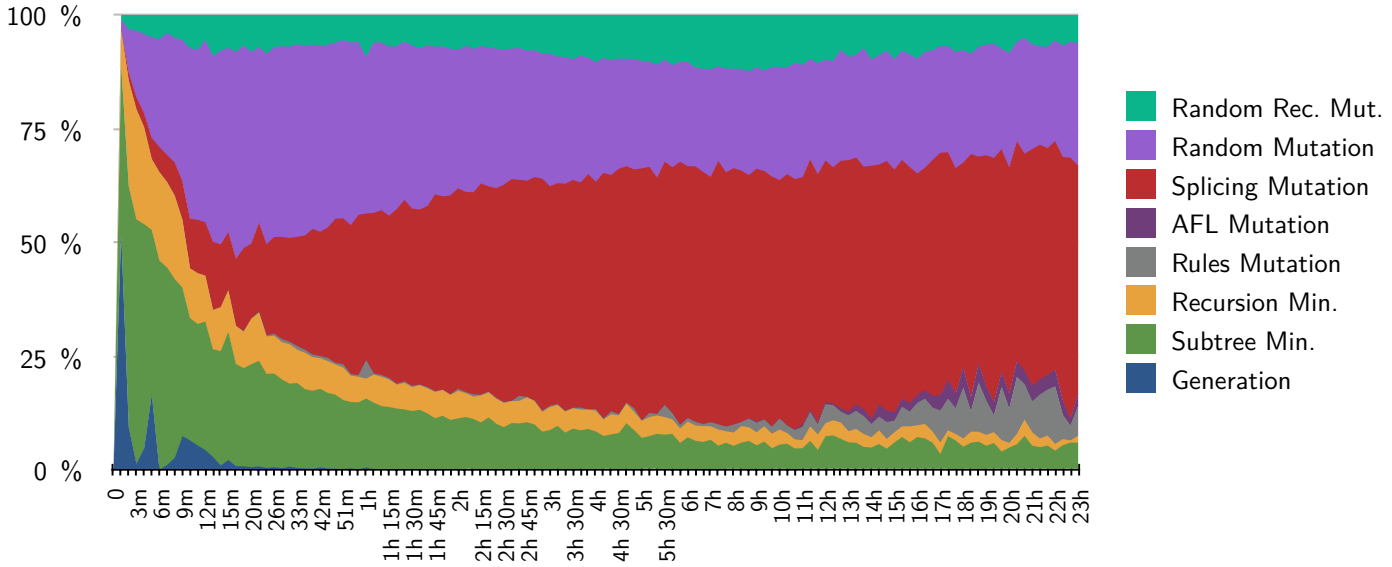


Fig. 5: Percentage of identified new paths for each mutation method, over 20 runs on each target.

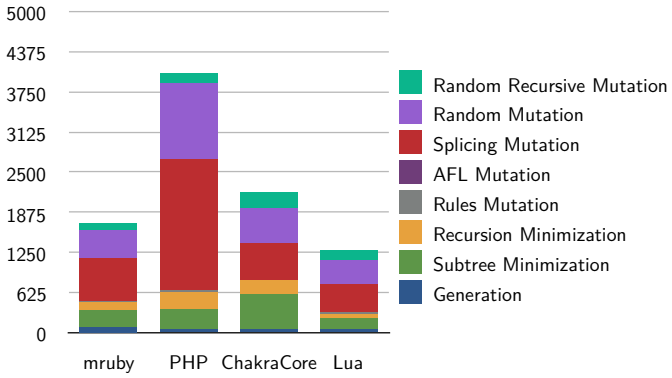


Fig. 6: Inputs that triggered new transitions for each target, grouped by generation/mutation method, for four specific runs.

interesting code fragments becomes by far the most effective mutation technique.

A similar behavior can be observed for the rules mutation. This mutation is only used after the minimization is done, and therefore it starts finding new paths only later in the fuzzing process. The generation and minimization methods find many new paths at the beginning, but after a couple of hours the splicing and random mutation make up more than 50% of the new identified paths. The Random Recursive Mutation finds less paths than the other mutations, but it finds paths that no other of our mutation methods can find: the PHP stack overflow vulnerability and two vulnerabilities of mruby (CVE-2018-10191 and CVE-2018-12248) have been found only by the Random Recursive Mutation.

[4]Skyfire is not an actual fuzzer, only a seed generator.

VII. Related Work

In the following, we discuss fuzzing approaches based on mutation or generation, where the latter are conceptually closer to NAUTILUS. Hence, we explain commonalities and differences of generation-based approaches in more detail. Table IV provides an overview of characteristics of most relevant existing approaches.

A. Mutation-Based Approaches

Mutation-based fuzzing has been a popular way to quickly find bugs, especially in input parsing. In contrast to generation-based fuzzing, only a test corpus is needed. Many of these approaches are based on AFL [18], a fuzzer that, while also supporting brute force, leverages genetic input mutation, guided by unique code coverage (counting only yet unseen execution paths). AFL is still popular, as it continues to beat competing fuzzers because of its sheer analysis cycle speed. However, it lacks syntactic insight for input generation, thus, paths guarded by complex syntactic or semantic checks remain unexplored. This is what other approaches try to solve by adding an interacting module with higher syntactic insight. Taint-based fuzzers like BuzzFuzz [29] or TaintScope [50] try to increase insight by leveraging taint tracing to map input bytes to function arguments or branch checks [24], [29], [43], [50]. This allows them to reduce input bytes that need to be mutated. However, taint-based mutations may still be syntactically (and even worse semantically) incorrect. NAUTILUS instead generates syntactically and semantically correct inputs. Instead of using (only) a taint-based companion module, there are also approaches that leverage computation-intensive symbolic execution that relies on constraint solving [22], [31], [32]. Because of its complexity, many approaches use symbolic execution only if is inevitable. For example, Dowser [35] only concentrates on interesting regions, i.e. loops with complex array accesses, and uses dynamic taint analysis to trace input

Fuzzer	Input Generation	Guided Fuzzing	Works w/o corpus	Bypasses input parsing	Bypasses semantic checks	Generally applicable
Radamsa	Corpus	✗	✗	✗	✗	✓
AFL	Mutation	✓	✗	✗	✗	✓
CSmith	CFG	✗	✓	✓	✗	✗
LangFuzz	Generation (corpus)	✗	✗	✓	✗	✓
IFuzzer	CFG + Corpus	✗	✗	✓	✓	✗
Skyfire	CFG + Corpus[4]	✗	✗	✓	✓	✓
NAUTILUS	loose CFG	✓	✓	✓	✓	✓

TABLE IV: Comparison of important related approaches.

bytes that map to these accesses. These bytes are analyzed symbolically, while bytes are treated as concrete values. Dowser's symbolic analysis is more likely to produce well-formed inputs, however, its limitation to buffer overflows prevents widespread use. In contrast, NAUTILUS can find arbitrary crashes, and can focus on a certain aspect of a program by adjusting the grammar accordingly. Driller combines aspects of all aforementioned mutation-based fuzzing approaches by leveraging dynamic and concolic execution. Driller [46] uses directed fuzzing until it is not able to generate new paths. Then, the concrete fuzzing input is passed to the symbolic execution engine that explores new paths that the fuzzer can continue to analyze. In comparison to NAUTILUS, Driller needs expensive symbolic execution to continue, while still not being able to easily generate semantically correct inputs.

B. Generation-Based Approaches

Generation-based fuzzers leverage either a grammar (or model), a corpus, or both to generate highly-structured, syntactically correct input. This is useful to analyze file viewers (like media players), interpreters, compilers, or e.g. XML parsers. While there are several general-purpose generation-based fuzzers [10], [41], [48], many approaches directly target a specific use case: CSmith [51] for C, LangFuzz [36] and IFuzzer [47] for JavaScript interpreters, and many more [7], [14], [20], [28], [44], [45]. In contrast, NAUTILUS is versatile and can be used to fuzz any application where the source code is available. In the following, we take a deeper look into important representatives of aforementioned categories.

CSmith [51] generates randomized test cases for C compilers based on a grammar. This grammar derives a subset of C programs by randomly applying production rules that avoids undefined/unspecified behavior in the C standard. CSmith, like NAUTILUS, is able to work without a corpus. Yet, it just derives inputs randomly, whereas NAUTILUS uses mutations based on subtrees to generate diverse input that allows a path to be further explored.

Radamsa [48] uses corpus to derive a CFG to represent its structure, then generates new inputs derived by this grammar. It also applies mutation to generate more diverse inputs: *global mutations* mutate the CFG, while *point mutations* are applied during input derivation. CFG creation as well as mutation may however introduce semantic errors. NAUTILUS directly leverages grammars that are for example provided by the ANTLR project [6], hence, can bypass semantic checks easily. Moreover, subtree mutations ensure that this is always the case.

LangFuzz [36] and IFuzzer [47] leverage a provided (context-free) language grammar to extract code fragments from a corpus. These code fragments are recombined to new inputs. In contrast to LangFuzz, IFuzzer uses genetic programming with a fitness function (for diversity) to generate more uncommon, but syntactically and semantically valid input. Skyfire [49] is a seed generator that uses a grammar and a corpus. The samples from the corpus are parsed (using the grammar) to get selection probabilities for each production rule. Then, low-probability rules are preferably used to derive uncommon seeds. Leaves in seeds' parsing tree representation are then replaced with other terminals that can be generated by the same rule. These seeds can then be used by fuzzers like AFL.

The key aspect of the aforementioned fuzzers is grammar-based recombination of samples to get uncommon, syntactically and hopefully semantically correct input (see again Table IV for an overview). NAUTILUS does not rely on a corpus that may already encode what behavior is interesting. Instead, it generates and recombines inputs guided by coverage feedback. This allows NAUTILUS to make use of its growing internal storage of (mostly) semantically correct inputs to greatly increase the chance of producing new additional inputs that are also semantically correct. In addition, NAUTILUS integrates techniques from mutation-based fuzzing: coverage feedback guides mutation and derivation of inputs. This allows NAUTILUS to find interesting inputs without relying on a corpus.

Additionally, some research has been conducted in the field of automatic grammar generation for fuzzing. Godefroid et al. [34] use neural networks to construct PDF grammar partially (limited to non-binary PDF data objects). Another approach by Godefroid et al. [30] leverages SMT solvers to generate a grammar. Similarly, Bastani et al. [23] implemented a grammar synthesis tool based on an oracle (the target program). Lastly, AUTOGRAM [37] automatically learns grammars from Java code, however, the approach does not seem to be adaptable to binary-only targets easily. These techniques might further simplify generation of grammars used in future fuzzing runs.

VIII. LIMITATIONS

NAUTILUS is significantly faster and more flexible than current alternative approaches, yet it has some limitations

that we discuss in the following. Similar to AFL and related tools, it needs source level access to add the instrumentations needed for coverage feedback. However, the methods described themselves could just as easily be implemented on top of Dynamic Binary Instrumentation [12], [43] or feedback mechanisms based on Intel PT [45]. Other grammar-based fuzzers typically require both a grammar as well as a set of good inputs that can be parsed with this grammar. NAUTILUS reduces this limitation, but still requires a grammar. Additionally, for maximum efficiency the grammar needs to contain a list of important symbols such as identifiers or class names. Lastly, while the scripting support is a very powerful primitive that can generate a multitude of non-context-free constructs, there are some common features (mostly file offsets) that sometimes require a significant restructuring of the grammar. While these are nontrivial issues, the next section details how they can be fixed using existent techniques.

IX. FUTURE WORK

To further ease the use of our fuzzer, one could switch the instrumentation-based backend with the AFL-QEMU mode backend. Then even binary targets that use highly structured input languages can be fuzzed effectively. When extending the grammars by important symbols, we manually added the output of the strings utility to the grammar. This step could be easily automated to further reduce the amount of manual work needed. Additionally, the dependence on a grammar can be drastically reduced by techniques that automatically infer a grammar from the program itself such as the tools proposed by Höschel et al. [37] or Bastani et al. [23], or by using machine learning techniques [33]. While our scriptable grammars are able to generate any decidable language, some common language features need complex scripts. For example, file offsets are hard to implement as the exact offset of a given substring is typically not known at the time of script execution. Thus, the script for the start rule needs to manually compute all offsets. Adding support for labels would probably ease the process of writing grammars for binary file formats. There are also interesting research directions that might increase fuzzing efficiency even more. As an example, other generation methods might be developed that perform better than the naive approach.

X. CONCLUSION

This work confirms that the use of grammars increases the effectiveness of fuzzing programs that interpret complex input. Combining grammar fuzzing with the use of instrumentation feedback improves the fuzzing process even more. Typically, adding feedback to grammar-based fuzzing increases the code coverage by at least a factor of two for our four targets: mruby, PHP, Lua, and ChakraCore. When comparing against tools not based on grammars, such as AFL, that only employ feedback driven fuzzing, the improvements over the seed corpus are even more drastic: In many cases we find more than ten times as much new coverage. Our results show that it is the combination of grammars and instrumentation that leads to this significantly increase in performance. This combination allows the fuzzer to automatically identify and recombine semantically valid fragments of code to drastically increase the performance. Additionally, we were found and reported thirteen new bugs in those four targets and received 2,600 USD in bug bounties.

ACKNOWLEDGMENTS

This work was supported by Intel as part of the Intel Collaborative Research Institute "Collaborative Autonomous & Resilient Systems" (ICRI-CARS). This work was co-funded by the DFG (projects P3 and S2 within CRC 1119 CROSSING, and HWSec), by the German Federal Ministry of Education and Research (BMBF, projects HWSec and iBlockchain) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP. The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 786669. The content of this document reflect the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains. Finally, we would like to thank Joel Frank for his valuable feedback.

REFERENCES

[1] About the antlr parser generator [online]. `http://www.antlr.org/about.html`. Accessed: 2018-04-17.

[2] Chakracore is the core part of the chakra javascript engine that powers microsoft edge [online]. `https://github.com/Microsoft/ChakraCore`. Accessed: 2018-06-13.

[3] Clang's source-base code coverage [online]. `http://releases.llvm.org/6.0.0/tools/clang/docs/SourceBasedCodeCoverage.html`. Accessed: 2018-07-11.

[4] gcov [online]. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Accessed: 2018-07-11.

[5] gramfuzz is a grammar-based fuzzer that lets one define complex grammars to generate text and binary data formats. [online]. `https://github.com/d0c-s4vage/gramfuzz`. Accessed: 2018-06-11.

[6] Grammars written for antlr v4 [online]. `https://github.com/antlr/grammars-v4`. Accessed: 2018-04-17.

[7] mangleme. Accessed: 2018-08-03.

[8] mruby. `http://mruby.org`. Accessed: 2018-06-13.

[9] mruby/mruby: Lightweight ruby [online]. `https://github.com/mruby/mruby`. Accessed: 2018-06-13.

[10] Peach fuzzer: Discover unknown vulnerabilities. [online]. `https://www.peach.tech/`. Accessed: 2018-07-10.

[11] Php: Hypertext preprocessor [online]. `http://php.net/`. Accessed: 2018-06-13.

[12] Pin - a dynamic binary instrumentation tool [online]. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`. Accessed: 2018-07-03.

[13] The programming language lua [online]. `https://www.lua.org/`. Accessed: 2018-06-13.

[14] PROTOS. `http://www.ee.oulu.fi/research/ouspg/protos`. Accessed: 2018-08-03.

[15] Security oriented fuzzer with powerful analysis options. `https://github.com/google/honggfuzz`. Accessed: 2018-08-07.

[16] shopify-scripts: Bug bounty program on hackerone. `https://hackerone.com/shopify-scripts/`. Accessed: 2018-06-13.

[17] Sulley: A pure-python fully automated and unattended fuzzing framework. [online]. `https://github.com/OpenRCE/sulley`. Accessed: 2018-06-11.

[18] Technical "whitepaper" for afl-fuzz [online]. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: 2018-06-12.

[19] *american fuzzy loop*. https://github.com/mirrorer/afl, 2017.

[20] *syzkaller: Linux syscall fuzzer.* https://github.com/google/syzkaller, 2017.

[21] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[22] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.

[23] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[24] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 818–825. IEEE, 2012.

[25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[26] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008.

[27] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[28] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 725–730. ACM, 2014.

[29] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[30] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[31] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[32] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[33] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[34] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

[35] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[36] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.

[37] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[38] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`. Accessed: 2018-08-03.

[39] Bruce McKenzie. Generating strings at random from a context free grammar. 1997.

[40] Travis Oliphant et al. NumPy: Open source scientific tools for Python. `http://www.numpy.org/`. Accessed: 2018-08-03.

[41] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 543–553. IEEE, 2016.

[42] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, Sep 1972.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[44] Jesse Ruderman. Introducing jsfunfuzz. *URL http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz*, 2007.

[45] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. 2017.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[47] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[48] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with model inference assisted fuzzing. *WOOT*, 2:1–2, 2008.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 579–594. IEEE, 2017.

[50] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[52] Zoltán Ésik and Szabolcs Iván. Büchi context-free languages. *Theoretical Computer Science*, 412(8):805–821, 2011.