Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis

Xuan-Bach D. Le[1], Corina Pasareanu[2,3], Rohan Padhye[4], David Lo[5], Willem Visser[6], and Koushik Sen[4]

[1]The University of Melbourne, Australia, bach.le@unimelb.edu.au

[2]Carnegie Mellon University, USA, corina.pasareanu@west.cmu.edu

[3]NASA Ames Research Center, USA, Corina.S.Pasareanu@nasa.gov

[4]University of California, Berkeley, USA, {rohanpadhye,ksen}@cs.berkeley.edu

[5]Singapore Management University, Singapore, davidlo@smu.edu.sg

[6]Stellenbosch University, wvisser@cs.sun.ac.za

DOI: 10.1145/3364452.3364455
http://doi.acm.org/10.1145/3364452.3364455

ABSTRACT

Fuzz testing has been gaining ground recently with substantial efforts devoted to the area. Typically, fuzzers take a set of seed inputs and leverage random mutations to continually improve the inputs with respect to a *cost*, e.g. program code coverage, to discover vulnerabilities or bugs. Following this methodology, fuzzers are very good at generating *unstructured* inputs that achieve high coverage. However fuzzers are less effective when the inputs are structured, say they conform to an input grammar. Due to the nature of random mutations, the overwhelming abundance of inputs generated by this common fuzzing practice often adversely hinders the effectiveness and efficiency of fuzzers on grammar-aware applications. The problem of testing becomes even harder, when the goal is not only to achieve increased code coverage, but also to find complex vulnerabilities related to other cost measures, say high resource consumption in an application.

We propose SAFFRON an adaptive grammar-based fuzzing approach to effectively and efficiently generate inputs that expose expensive executions in programs. SAFFRON takes as input a user-provided grammar, which describes the input space of the program under analysis, and uses it to generate test inputs. SAFFRON assumes that the grammar description is *approximate* since precisely describing the input program space is often difficult as a program may accept unintended inputs due to e.g., errors in parsing. Yet these inputs may reveal worst-case complexity vulnerabilities. The novelty of SAFFRON is then twofold: (1) Given the user-provided grammar, SAFFRON attempts to discover whether the program accepts unexpected inputs outside of the provided grammar, and if so, it repairs the grammar via grammar mutations. The repaired grammar serves as a specification of the actual inputs accepted by the application. (2) Based on the refined grammar, it generates concrete test inputs. It starts by treating every production rule in the grammar with equal probability of being used for generating concrete inputs. It then adaptively refines the probabilities along the way by increasing the probabilities for rules that have been used to generate inputs that improve a cost, e.g., code coverage or arbitrary user-defined cost. Evaluation results show that SAFFRON significantly outperforms state-of-the-art baselines.

1. INTRODUCTION

Algorithmic worst-case complexity of software has long been an important problem. Software vulnerabilities or performance degradation resulted from worst-case executions of programs can be exploited to launch Denial-of-Service (DoS) attacks, bringing down entire websites [1], disabling or bypassing web application firewalls [2], or consuming a huge amount of CPU resources by simply performing hash-table insertions [3, 4]. The results of these attacks are often costly, and thus, it is crucial to identify issues related to worst-case complexity beforehand.

Several solutions to reason about software's worst-case complexity have been proposed and shown pragmatic results, e.g., [10, 12, 13, 22, 26, 27]. However, they still have their own weaknesses. Approaches that investigate theoretical analyses such as [10,12,13,27] often have scalability issue, while more practical and scalable solutions such as fuzzing [22, 26] suffer from inefficiency, i.e., they take a long time to generate desired test cases. This is especially true when dealing with applications that take complex structured inputs, for instance inputs that can be specified by a *grammar*. Fuzzers typically generate a large number of unstructured inputs using random mutations (bit/byte flips) and continually evolve the inputs according to a cost function, such as program code coverage [8] or resource usage [22, 26]. Due to the nature of random mutations, the overwhelming abundance of inputs generated by this common fuzzing practice often adversely hinders the efficiency and effectiveness of fuzzers, especially on applications whose inputs conform to a grammar. That is, the majority of unstructured inputs generated by fuzzers are often unduly rejected early during the parsing phase by grammar-aware applications, and hence, fail to reveal worst-case executions of those programs. On the other hand, existing grammar-based fuzzers [11, 28] aim to only increase coverage and are thus not tailored for worst-case analysis.

In this paper, we propose SAFFRON – an adaptive grammar-based fuzzing approach that aims to generate inputs that expose worst-case executions of grammar-aware applications. SAFFRON complements existing techniques by taking a radical approach to grammar-based fuzzing. Given a user-provided grammar – which is an approximate specification of the input space admissible by the program under test, SAFFRON evolves the grammar and subsequently uses the refined grammar to generate inputs. SAFFRON assumes that the user-provided grammar is *approximate*. This is because in practice a program may accept unintended inputs due to e.g. errors in parsing, yet those inputs may reveal worst-case behaviors. This danger is real as for instance processing modules for common formats such as JSON or XML have been found vulnerable with respect to maliciously constructed data [5].

SAFFRON works in two main phases: *grammar refinement* and *adaptive grammar-based input generation.* SAFFRON first attempts to discover whether the grammar correctly describes the actual admissible input space of the program, and if not, it *repairs* the grammar accordingly via a set of grammar mutations. SAFFRON then uses the repaired grammar for the input generation phase. In lieu of random selection of production rules from the grammar to generate inputs, SAFFRON takes an adaptive approach to choose production rules more wisely. That is, it starts by treating every production rule with equal probability to be chosen. It then adjusts the probability over time by increasing the probability for rules that have been used to generate inputs that improve a cost, e.g., program code coverage or a cost describing a resource consumption (such as execution time). Overall, the co-evolution of the grammar's production rules (via grammar refinement) and the probability of production rules (via adaptive input generation) is the key novelty that empowers SAFFRON's efficiency and effectiveness with respect to finding worst-cost executions in a program.

We compare SAFFRON against a random grammar-based fuzzer and PERFFUZZ – a very recent state-of-the-art fuzzer that targets worst-case inputs [21]. We performed experiments on five Java subject programs in the DARPA cybersecurity challenges [6]. Experiment results are encouraging: SAFFRON is twice to eight times better than the baselines. More interestingly, SAFFRON reveals a vulnerability that the baselines are unable to discover.

The rest of the paper is organized as follow. Section 2 presents an example to motivate our approach. Section 3 explains our framework, followed by Section 4 that describes the evaluations of our approach against baselines. Section 5 presents related work. Section 6 concludes and presents future work.

2. MOTIVATING EXAMPLE

We present a motivating example, which is taken from recent DARPA engagements [6]. It is a calculator program that computes the value of a given mathematical expression that involves addition, subtraction, multiplication, division, exponentiation, and root $(+, -, *, /, \hat{}, r)$. The program, which contains approximately 6000 lines of code, is designed to handle exceptionally large numbers and thus consists of complex algorithms that involve several nested looping constructs. The use of these nested loops makes it hard for either manual or automated analysis techniques. The calculator accepts inputs of different forms such as large *numerical* or *roman* numbers. It terminates outright if the calculation results in a too-large number or some exceptions happen. The challenge question is then: *Can an attacker derive an input that renders the calculator to consume memory or runtime over a predefined budget?* An attack, if can be synthesized, can be used to inform developers of the application on security problems and fix it before the application is deployed. Note that, synthesizing such attack is feasible, yet challenging because: (1) the grammar of the inputs that the calculator actually accepts is not known in advance, and (2) the calculator terminates early without much resource consumption if the given input is inadmissible by the calculator.

To compose such an attack, a naive attacker can just make initial guesses on what forms of inputs that the calculator may accept, e.g., prefix expressions such as (+ 3 4), infix expressions such as (3 + 4), etc. Traditional fuzzers such as AFL [8] can then be employed to generate more random inputs based on the seed inputs given by the attacker's initial guesses. This approach, however, is ineffective since such fuzzers often generate an overwhelmingly large number of unstructured inputs which can be unduly rejected early by the application without much resource consumption.

```
grammar Calculator;
INT   : [0-9]+;
WS    : [ \t\r]+ →skip;
input : exp EOF ;
exp : exp '+' exp | exp '-' exp
    | exp '*' exp | exp '/' exp
    | exp '^' exp | exp 'r' exp
    | '(' exp ')' | INT ;
```

Figure 1: Grammar `Calculator` for infix expressions.

A seemingly more effective solution to derive an attack could be that the attacker first writes a grammar that expresses her/his *anticipation* on the input space that the application may accept, e.g., grammar for infix expressions as depicted in Figure 1. This grammar can then be used by a random grammar-based fuzzer which randomly generates inputs conforming to the grammar. However, this approach, despite being able to generate more valid inputs, still cannot generate a desired attack as we confirmed empirically. What could possibly hinder the effectiveness of such supposedly judicious approach? The answer lies in the user-provided grammar and how it is used to generate inputs.

First, user's anticipation on admissible input space of the application, as expressed via a user-provided grammar, may not be accurate nor complete. Also, in lieu of random selection of production rules for input generation, the rules should be chosen more wisely. That is, production rules that can be used to generate inputs that consume more resources, e.g., running time, should be favored. For example, input $(123\ r\ 3)$ can be more expensive to compute than input $(123 + 3)$ due to the use of root versus addition. To realize these ideas, our approach SAFFRON enhances the random grammar-based approach by employing two steps: *grammar refinement* and *adaptive grammar-based fuzzing.*

Our approach SAFFRON first uses a grammar refinement phase to discover that the user-provided grammar actually under-approximates the actual input space accepted by the calculator. That is, the calculator indeed accepts inputs outside the grammar that users expect the application to accept. SAFFRON then *refines* the grammar accordingly to better capture the application's input space. In this example, suppose that user anticipates the input space of the calculator as grammar `Calculator` depicted in Figure 1. SAFFRON discovers that the calculator indeed accepts a grammar `Calculator`$^+$ that generalizes the grammar `Calculator`. `Calculator`$^+$ accepts inputs of the form `(LPAR exp RPAR)`$^+$. Concretely, the application accepts inputs such as $(2+3)(4*5)(9-12)$ that users do not expect. SAFFRON subsequently uses an *adaptive* grammar-based fuzzing, which judiciously *favors* production rules that have been used to generate more expensive inputs. In short, the interplay of the two phases, i.e., grammar refinement and adaptive grammar-based fuzzing, enables SAFFRON to generate costly inputs more efficiently and effectively. Based on the repaired grammar, SAFFRON adaptively finds an input that exposes a trace that has cost twice as large as inputs found with competing techniques, for the same input budget.

3. OVERALL FRAMEWORK

SAFFRON takes as input a user-provided grammar, refines the grammar, and then generates inputs based on the refined gram-

$$\begin{tikzcd}
\mathcal{G} \arrow[r] & \begin{array}{c}\mathcal{I} = \{i_1, i_2, ...\},\\ \text{where } i_j \in \mathcal{G}\end{array} \arrow[r, "\textsf{AFL}"] & \begin{array}{c}\mathcal{I}^* = \{i_1^*, i_2^*, ...\},\\ \text{where } i_j^* \in \mathcal{P} \wedge i_j^* \notin \mathcal{G}\end{array} \arrow[d] \\
& & \mathcal{G}^*, \text{ where } i_j^* \in \mathcal{I}^* \wedge i_j^* \in \mathcal{G}^*
\end{tikzcd}$$

Figure 2: Grammar Refinement in Saffron

$$\begin{tikzcd}
\mathcal{G}^*_{prob} = \{(\mathcal{R}_1, p_1), (\mathcal{R}_2, p_2), ...\} \arrow[r] & \begin{array}{c}\mathcal{I} = \{i_1, i_2, ...\},\\ \text{where } i_j \in \mathcal{G}^*_{prob}\\ \wedge\, \mathcal{C}^{\mathcal{P}}_{i_j} > \mathcal{C}_{max}\end{array} \arrow[l, bend left, "update(\mathcal{C}_{max}, p_i)"]
\end{tikzcd}$$

Figure 3: Adaptive Grammar-based Fuzzing in Saffron

mar. It thus has two phases: *grammar refinement* (Figure 2) and *adaptive input generation* (Figure 3). We describe each phase of the framework below.

3.1 Phase I: Grammar refinement.

The goal of this phase is to refine the user-provided grammar to better capture the program's actual admissible input space. To achieve this goal, SAFFRON first attempts to discover unexpected inputs that are accepted by the program but fall outside of the given grammar representing user's anticipation on input space that the program should accept. Next, SAFFRON attempts to repair the grammar, rendering the discovered unexpected inputs (if any) to be admissible. We describe these steps below.

Discovery of unexpected inputs. Overall, this step tries to discover a set of inputs that witnesses the behaviors that users do not expect a program under analysis $\mathcal{P}$ to hold. Given a user-provided grammar $\mathcal{G}$ that expresses user's intention on the input space that $\mathcal{P}$ should accept, SAFFRON randomly generates a set of inputs $\mathcal{I}$ admissible by $\mathcal{G}$. It then explores the input space $\mathcal{I}^*$ nearby $\mathcal{I}$ by allowing coverage-guided fuzzing such as AFL [8] to randomly mutate inputs in $\mathcal{I}$. Note that AFL-like fuzzing techniques are typically very good at generating unstructured inputs that are likely to violate an underlying grammar. SAFFRON thus takes advantages of this fact and employs AFL to generate inputs $\mathcal{I}^*$ that are not admissible by $\mathcal{G}$, but unexpectedly accepted by the program under analysis $\mathcal{P}$. In other word, $\mathcal{I}^*$ witnesses the behaviors that users did not anticipate the program $\mathcal{P}$ to have.

Grammar repair. Given $\mathcal{I}^*$ containing inputs admissible and inadmissible by program P and grammar $\mathcal{G}$ respectively, SAFFRON attempts to derive a grammar $\mathcal{G}^*$ that accepts $\mathcal{I}^*$. Compared to the user-provided grammar $\mathcal{G}$, $\mathcal{G}^*$ thus serves as a better approximation for the actual input space accepted by program $\mathcal{P}$. Note that when evaluating $\mathcal{I}^*$ on $\mathcal{G}$, we can derive a set of locations $\mathcal{L}$ in $\mathcal{G}$ that possibly renders $\mathcal{I}^*$ inadmissible. To derive $\mathcal{G}^*$, SAFFRON continually applies a number of grammar-based mutation operators on locations $\mathcal{L}$ in $\mathcal{G}$ to create a set of candidate grammars $\{\mathcal{G}_i\}$. We adopted the mutation operators proposed by Offutt et al. [23]. Next, each candidate grammar in $\{\mathcal{G}_i\}$ is evaluated against $\mathcal{I}^*$. Candidates are then ranked by the number of inputs in $\mathcal{I}^*$ that they accept. The best candidate is then chosen to be further mutated. This process is repeated until $\mathcal{G}^*$ that accepts all inputs in $\mathcal{I}^*$ is found or a number of iterations is reached. Recall that our main goal is not to find a perfect grammar that precisely reflects the input space of program $\mathcal{P}$. Instead, any grammar accepted by $\mathcal{P}$, that is unexpected by users, is of our interest.

3.2 Phase II: Adaptive grammar-based inputs generation.

This phase takes as input the refined grammar $\mathcal{G}^*$ discovered in previous phase, and generates inputs conforming to $\mathcal{G}^*$. This process is parameterized by a depth d that limits how deep the grammar can be explored. Let us now informally define $\mathcal{G}^*$ and then explain the intuition behind the design of this phase. We will then explain the algorithm in details.

Rationale and intuition of algorithm design. Let us first simply consider $\mathcal{G}^*$ as a set of production rules $\{\mathcal{R}_i\}$. To generate inputs conforming to $\mathcal{G}^*$, one can just naively choose $\mathcal{R}_i$ at random to explore the grammar until a terminal node and the depth limit d are reached. However, this random approach may not be effective at generating inputs that can expose worst-case complexity, e.g., running time. Our intuition is that not every production rule is equally useful for generating worst-case inputs. Some rules can indeed be used to generate inputs that are more expensive, e.g., cause program $\mathcal{P}$ a longer running time, than inputs generated by using other rules.

We thus propose to *favor* those rules that lead to inputs that increase a cost when exploring the grammar. To achieve this, SAFFRON assigns a probability p_i to account for the likelihood of each production rule $\mathcal{R}_i$ to be used for generating inputs. Over time, SAFFRON refines this probability by increasing p_i according to the number of times the rule $\mathcal{R}_i$ have been used to generate costly inputs, e.g., inputs rendering the program under analysis $\mathcal{P}$ to run longer or to consume more resources. The higher the probability p_i is, the more likely that the corresponding rule $\mathcal{R}_i$ can be used to generate inputs.

Algorithm in details. Let us now explain the algorithm in more details. This phase keeps track of a maximum cost $\mathcal{C}_{max}$ by the best input generated so far, e.g., input that causes program $\mathcal{P}$ longest running time. It tries to increase this cost $\mathcal{C}_{max}$ over several iterations. In each iteration, it generates a number of inputs based on the given grammar and then selects top n inputs that improve the maximum cost $\mathcal{C}_{max}$ of the previous iteration. These selected inputs are then used to update the probability of grammar rules that have been used to generate them. The probability of a grammar rule is accounted by the number of times the rule has been used to generate costly inputs. The updated probability renders the algorithm adaptive, i.e., over time, rules that accumulate higher probabilities are more likely to be used for input generation in the next iteration. The whole process is repeated several times and ultimately outputs the best input that it generated so far in terms of cost.

At the core of our algorithm, the most crucial step is how input generation can be guided by the probability of production rules, which is adaptively adjusted over time. Algorithm 1 gives more details about this step. Particularly, the grammar is traversed starting from a rule name until a depth d is reached (Line 9). Production rules associated with the given rule name are then identified (Line 11). Among these rules, a tournament selection based on the rules' probability is then used to choose the rule to explore next (Line 17). Note that this selection favors rules with higher probability. Once a rule is elected, all elements of the rule will be visited. If an element is a nonterminal, i.e., a rule name, the traversal procedure is recursively called on that element with depth limit decreased by one (Line 24). If the depth limit is exceeded (Line 13), the algorithm allows backtracking and then chooses a different rule to explore.

Algorithm 1 SAFFRON's cost-guided grammar-based input generation algorithm

```
Input:
    $\mathcal{G}$                                   ▷ Grammar to explore
    $d$                                   ▷ Depth limit to explore a grammar
    $b$                                   ▷ A loop bound
    $r$                                   ▷ Rule name to start exploring the grammar
Output:
    Set of inputs generated
function InputGeneration($\mathcal{G}, d, b, r$)
    $\mathcal{I} \leftarrow \{\}$
    for $k \leftarrow 0$ to b do
        $i \leftarrow RecursiveGen(\mathcal{G}, d, r)$
        $\mathcal{I} \leftarrow \mathcal{I} \cup i$
    end for
    return $\mathcal{I}$
end function
function RecursiveGen($\mathcal{G}, d, r$)
    $\mathcal{G} \leftarrow \{(r_j \rightarrow \{(\mathcal{R}_i^{r_j}, p_i^{r_j})\})\}$     ▷ Let $\mathcal{G}$ be a set
    of mapping from each rule name $r_j$ to a set of tuples of (rule
    $\mathcal{R}_i^{r_j}$, probability $p_i^{r_j}$). $r_0$ denotes root rule name.
    $rRules \leftarrow \{(\mathcal{R}_i^r, p_i^r)\}$     ▷ By finding $r$ in $\mathcal{G}$
    $result \leftarrow None$
    if $d < 0$ then
        return $result$
    end if
    while $rRules.size > 0 \wedge result == None$ do
        $chosenRule \leftarrow TournamentSelection(rRules)$     ▷
    Selection guided by $p_i^r$. Higher $p_i^r$, more likely $\mathcal{R}_i^r$ be chosen
        $rRules \leftarrow rRules \cap chosenRule$
        for all $node \in chosenRule$ do
            if $node$ is terminal then
                $result \leftarrow result \mathbin{++} node$
            else                                   ▷ node is nonterminal
                $rN \leftarrow node.GetName()$
                result $\leftarrow RecursiveGen(\mathcal{G}, d-1, b, rN)$
            end if
        end for
    end while
end function
```

4. IMPLEMENTATION & EVALUATION

We implemented SAFFRON using ANTLR4 [25], which allows SAFFRON to generically work on grammars written in ANTLR. We compared SAFFRON against two baselines: (1) A *random grammar-based fuzzing* approach, denoted as GRAMRAND. GRAMRAND's design basically follows our framework SAFFRON. However, GRAMRAND does not favor production rules like our approach, but instead randomly selects production rules for generating concrete inputs, and (2) PERFFUZZ – a recent state-of-the-art fuzzer for generating worst-case inputs [21]. Note that PERFFUZZ does not exploit input structure, but instead, treats inputs as sequence of bytes. We could not compare directly with a grammar-based fuzzer that generate worst-case inputs since to our knowledge, our approach is the first to directly exploit input grammar for generating inputs that expose performance problems.

Subject Programs and Experiment Settings. We experimented with Java subject programs from DARPA engagements [6], which include four implementations of Calculator and a Python Static Analysis (PSA) tool. These programs, whose sizes range from 6600 to 8300 lines of code, are algorithmically nontrivial. The calculators are designed to handle exceptionally large numbers and thus contain sophisticated algorithms which involve abundance of complex looping constructs. The PSA is a static analyzer for a subset of the Python language. It takes as input a Python program, compiles and writes intermediate files in JSON format to physical disk to serve static analyses. For the calculators examples, we wrote the grammars ourselves, based on the description provided by DARPA, while for the PSA we used the JSON grammar in ANTLR [7]. On fuzzing these programs, we capped the size of the inputs to the programs at 10KB (following the challenge questions by DARPA). We run each tool against each subject program five times, each of which with a different seed and is timed out after 5 hours, and report best results. Experiments were conducted on a Intel(R) Core(TM) i7-6820HK CPU @ 2.70GHz, 2701 Mhz, 4 Core(s), 8 Logical Processor(s), and 8GB of RAM.

Table 1: Experiment Results

Programs	Eval. Metric	SAFFRON	GRAMRAND	PERFUZZ
Calculator 1	#Jumps	56,164,514	31,883,572	23,847,851
Calculator 2	#Jumps	45,510,716	32,626,634	21,670,056
Calculator 3	#Jumps	30,129,208	11,929,974	10,982,233
Calculator 4	#Jumps	39,501,218	18,736,256	12,181,585
PSA	File size (Kb)	33,347	4,420	N/A

Evaluation Metrics and Results. The challenge questions that we pose are: "can an input render the calculators to run longer than a predefined budget?", and "can an input cause the PSA to write a file larger than a predefined budget?". These questions translate to two evaluation metrics used for the calculators and the PSA respectively. The evaluation metric used for the calculators is the largest number of jump instructions (#Jumps) that an input can execute. The #Jumps commonly serves as a proxy for running time [21, 22]. The evaluation metric used for the PSA is the largest size of file that an input can render the PSA to write to disk. On both metrics, the higher the #Jumps and file size, the better.

Table 1 shows the experiment results. Overall, SAFFRON significantly outperforms both baselines. On the calculators, SAFFRON generates inputs that execute up to approximately three times more jump instructions than GRAMRAND and PERFFUZZ. We note that both SAFFRON and PERFFUZZ also generated inputs that cause the calculator 3 to hang, which are $(5-7-6-1000000)r8-1/1-1-1000000*3*2$ and $100-84184100-840000-500-840000-55rr2$ respectively. The results shown in Table 1 are for inputs that do not cause the programs to hang. On the PSA, SAFFRON generates an input that causes the PSA to write to disk a large file, which is approximately eight times larger than that by the RANDOM approach. More interestingly, SAFFRON-generated input reveals a real vulnerability which can crash the PSA's server down due to the large file written to disk. Note that PERFFUZZ cannot handle the PSA due to its lack of support in generating inputs that particularly optimize for size of file written out to physical disk.

5. RELATED WORK

Despite the abundance of fuzzers proposed recently, e.g., [11, 26, 28], only a few address worst-case analysis, e.g., PERFFUZZ [21] and SLOWFUZZ [26], which however, are not aware of input structure. We show experimentally that our grammar-based fuzzer outperforms state-of-the-art fuzzer PERFFUZZ. PERFFUZZ is originally designed to work on programs written in C programming languages but it has an extended interface that allows it to work on Java programs. We compare PERFFUZZ with our approach SAFFRON by using JQF [24], which is an interface of PERFFUZZ for Java programs. We could not compare with SlowFuzz since it does not work on Java programs. We assume that the developer has an idea of a starting grammar (or the application itself comes with a grammar) and we attempt to refine it, unlike recent works on grammar synthesis [9, 14] which synthesize grammars from scratch. Our work is also related to automated program repair, e.g., [16–20,29,30]. Similar to those works, we use mutations, but we focus on grammar refinement/repair for grammar-based fuzzing.

6. CONCLUSION AND FUTURE WORK

We presented Saffron – an adaptive grammar-based fuzzing approach to generate inputs that expose costly executions in programs. Based on an input grammar, Saffron adaptively selects production rules that can potentially be used to generate more costly inputs. Preliminary evaluations showed superior performance of Saffron over a state-of-the-art performance fuzzer on five subject programs from the recent DARPA challenges.

In the future, we will extend the evaluations to more subjects with larger size to further validate the current preliminary results. Also, it would be interesting to explore different ways to perform the probabilistic grammar-based fuzzing, e.g., learning a probabilistic automaton to predict which production rule should be chosen next after a previous rule. We also plan to extend our approach to generate test cases that can facilitate automated program repair as suggested in [15]. Overall, our work pragmatically complements the existing research in fuzzing and opens up interesting future directions.

Acknowledgement

This work was partially supported by NSF Grant CCF 1901136.

7. REFERENCES

[1] http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[2] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5021.

[3] https://www.phpclasses.org/blog/post/171-PHP-Vulnerability-May-Halt-Millions-of-Servers.html.

[4] https://meta.stackoverflow.com/questions/32837/why-does-stack-overflow-use-a-backtracking-regex-implementation.

[5] https://github.com/codehaus-plexus/plexus-util/issues/57.

[6] https://www.darpa.mil/program/space-time-analysis-for-cybersecurity.

[7] https://github.com/antlr/grammars-v4/blob/master/json/JSON.g4.

[8] American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, volume 52, pages 95–110. ACM, 2017.

[10] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *ACM SIGPLAN Notices*, pages 467–478, 2015.

[11] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. PLDI '08, pages 206–215. ACM, 2008.

[12] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. POPL '09, pages 127–139. ACM, 2009.

[13] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. POPL '17, pages 359–373, 2017.

[14] Matthias Höschele, Alexander Kampmann, and Andreas Zeller. Active learning of input grammars. *arXiv preprint arXiv:1708.08731*, 2017.

[15] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *41st International Conference on Software Engineering*, pages 524–535. IEEE Press, 2019.

[16] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. Jfix: semantics-based repair of java programs via symbolic pathfinder. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–379. ACM, 2017.

[17] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. ESEC/FSE, pages 593–604. ACM, 2017.

[18] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 428–432. IEEE, 2016.

[19] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[20] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[21] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *ISSTA '18*, pages 254–265. ACM, 2018.

[22] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *ISSTA '18*, pages 322–332. ACM, 2018.

[23] Jeff Offutt, Paul Ammann, and Lisa Liu. Mutation testing implements grammar-based testing. In *Mutation-ISSRE Workshops '06*, pages 12–12.

[24] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 398–401, New York, NY, USA, 2019. ACM.

[25] Terence Parr. *The definitive ANTLR 4 reference.* 2013.

[26] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. CCS '17, pages 2155–2168. ACM, 2017.

[27] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reason.*, 59(1):3–45, June 2017.

[28] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE*, 2018.

[29] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[30] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.