2018 42nd IEEE International Conference on Computer Software & Applications

Improving Fitness Function for Language Fuzzing with PCFG Model

Xiaoshan Sun[†], Yu Fu[†]
[†] *Trusted Computing and Information Assurance Laboratory*
Institute of Software, Chinese Academy of Sciences
Beijing, China
sunxs@tca.iscas.ac.cn, fuyu@tca.iscas.ac.cn

Yun Dong[‡]
[‡]*Beijing Capitek Co., Ltd.*
Beijing, China
dongyun@capitek.com.cn

Zhihao Liu[†], Yang Zhang[†]
mikeliu@tca.iscas.ac.cn
zhangyang@tca.iscas.ac.cn

***Abstract*—In this paper, we propose to use machine learning techniques to model the vagueness of bugs for language interpreters and develop a fitness function for the language fuzzing based on genetic programming. The basic idea is that bug-triggering scripts usually contain uncommon usages which are not likely used by programmers in daily developments. We capture the uncommonness by using the probabilistic context-free grammar model and the Markov model to compute the probabilities of scripts such that bug-triggering scripts will get lower probabilities and higher fitness values.**

We choose the ROC (Receiver Operating Characteristic) curves to evaluate the performance of fitness functions in identifying bug-triggering scripts from normal scripts. We use a large corpus of JavaScript scripts from Github and POC test cases of bug-reports from SpiderMonkey's bugzilla for evaluations. The ROC curves from the experiments show that our method can provide better ability to rank the bug triggering scripts in the top-K elements.

***Keywords*-language fuzzing, probabilistic context-free grammar, evolutionary algorithm, script ranking, Markov chain**

I. Introduction

Modern web browsers typically integrate interpreters to parse, compile, and execute scripts written in JavaScript. Prominent examples of such interpreters include the JavaScript engine V8 [1] of Google Chrome, SpiderMonkey [2] of Mozilla FireFox, and SquirrelFish [3] of Apple's Safari browser. Language fuzzing is one effective method to find security bugs in web browsers to improve their security as web browsers have been increasingly targeted by hackers.

All the possible scripts defined by the language under test form the script-space and the language fuzzing can be viewed as space search problems. Genetic Programming (GP) is one effective method to search large space for target solutions. It has been applied in testing [4] [5] [6] and fuzzing [7]. The bugs of interpreters are diverse and the concept of faults in the interpreters is vaguely defined, and therefore the fitness function cannot de defined directly. We propose to use the machine learning based method to define the fitness function for language fuzzing. Most bug-triggering scripts contain uncommon usages and this uncommonness can be modeled as low-probability of producing this script and the probability can be calculated by Probabilistic Context-Free Grammar(PCFG) as this model is used to measure the likely-hood of a tree and a script can be parsed into an Abstract Syntax Tree(AST). However, the longer scripts therefore will be considered less likely and it doesn't hold for our concept "bug-triggering" as usually such scripts are short. This means further design is required.

Contributions: The main contributions of this paper are:

- We propose to use the machine learning techniques to model the concept of "bug-triggering" in the interpreter fuzzing and our fitness function called Naturalness Heuristic for Fuzzing is based on the PCFG model and integrate it with the model using Markov chain model and the result is called MPCFG model. We use the uniform distribution to normalize the probabilities computed the MPCFG model and use the geometric mean to reduce the effect of script-length values. As a result the fitness values are almost independent of script lengths. Therefore we can alleviate the bloating problem in the GP(Genetic Programming) based language fuzzing.
- We performed an experiment to evaluate the effectiveness of fitness functions using a large collection of open source JavaScript scripts. The experiment shows that our method is more effective than those used in IFuzzer and pure PCFG model in distinguishing normal and uncommon scripts, which may suggest that our method can be used as a better fitness function for JavaScript fuzzers to generate bug-triggering scripts more efficiently.

II. Related Work

Language Fuzzing is in active research. Early work in language fuzzing, such as including LangFuzz [8], JsFunFuzz [9] and CSmith [10], mainly used manually crafted rules to guide the generation of test case programs. But they are different from each other in terms of how the program generation is guided, CSmith generates C programs cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs; LangFuzz [8] uses code mutation to generate new programs from previously generated ones that have triggered bugs in interpreters; JsFunFuzz [9] is an early fuzzer for the SpiderMonkey. JsFunFuzz and DOMFuzz have reportedly found 790 security critical bugs. However, these approaches may degenerate into random search [7].

Our work also relates to the GP based fuzzing. The fitness functions of evolutionary fuzzer are usually coverage based. DeMott et al. [11] propose a gray-box technique to generate new inputs with better code coverage to find bugs by using block coverage as the fitness value. Kifetew et al. [12] combines stochastic grammar with GP and determines the fitness value by running the system under test with all unparsed trees from the suite and measuring the amount of branches that are covered, as well as the distance from covering the uncovered branches. VUzzer [13] calculates fitness of each input as the weighted sum of the frequencies of executed basic blocks. They mainly use code coverage for fuzzing and their fitness function is defined independently of specific program. IFuzzer [7] the structural metrics such as the cyclomatic complexity for the program as the fitness value for scripts in the genetic programming.

Machine learning methods have been successful applied in statistical code completion [14], plagiarism detector based on n-gram model [15], bug detection based on n-gram language model [16], and likely buggy program behaviour detection using Bayesian reasoning [17]. The statistical basis of applying machine learning methods to program analysis tasks is naturalness of software, studied by Hinder [18] and Ray [19]. Machine learning based methods are applied in fuzzing as well. Godefroid et al. [20] proposed a method to generate new input using neural-network-based techniques. They sample the learned distribution of samples by choosing a value with probability lower than the threshold p_t. Skyfire [21] learns the probabilistic context-sensitive grammar to specify the syntax features and semantic rules, and generate seed inputs using the learned grammar. Skyfire prefers low-probability production rules during generation to produce uncommon-inputs. TreeFuzz [22] learns the probability distribution from code corpus using a PCFG-like model and generate new data from the models by creating trees in a depth-first manner.

Our work differs from related work in many aspects. We propose to use the PCFG and Markov model for the concept of "bug-triggering" triggering scripts and we propose a fitness function for GP based language fuzzing while previous work use code coverage based measurement or code complexity measurement. Longer scripts usually have lower probabilities in the PCFG-based model and as a result both longer common scripts and uncommon scripts have low probabilities and the model may fail to discriminate them. We develop a method to significantly reduce the influence of their lengths so that the model can discriminate the uncommon scripts from common scripts. We show that our model can serve as better fitness function to describe the vague concept of "bug-triggering" scripts without using the code complexity and code coverage methods and works better than code complexity based methods.

III. Naturalness of Software for Fuzzing

In this section we explain the basic idea of using the machine learning methods to model the concept of "bug-triggering" scripts to construct the fitness function for GP-based language fuzzing.

```
1 function testinput(re, str) {
2   var midstring;
3   if (re.test(str)) {
4     midstring = ' contains ';
5   } else {
6     midstring = ' does not contain ';
7   }
8   console.log(str + midstring + re.source);
9 }
```

Fig. 1. Typical usage example of `Regexp` from Mozilla web docs [23]

It has been observed by A. Hindle, et al. [18] that programming languages, in theory, are complex, flexible and powerful, but the programs that programmers actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical models for software engineering tasks.

This observation also holds in scripts written by humans and the "bug-triggering" scripts are intuitively bizarre in most cases and the reason is that the humans usually do not use the script language in this way. This is called uncommonness.

For example, Figure 1 shows a characteristic usage of "`Regexp`" in the JavaScript. The "`re.test(str)`"(Line 3) is the branch condition and the flowing statements in the branch continue to store the results. This is the kind of logic that the programmers will write in most JavaScript programs. Figure 2 gives an example script which violates this usage by updating the `input` (Line 7) and this script triggers a fault in one version of SpiderMonkey. This usage can be considered as uncommon. As uncommon scripts are unlikely and the usages may become "blind" points for developers and as a result the common usage reflect certain assumptions and this may cause the developers of interpreters to write code with these assumptions which can be invalid in certain uncommon scripts. So our idea is that learn the commonness from corpus of normal scripts developed by human programmers and the learned information can be used to compute the uncommonness of a script by measuring how far it deviates from common scripts. A script with larger deviation may be more likely to trigger bugs in interpreters. This deviation can be used to compute the fitness of scripts for language fuzzing based on the GP.

IV. Design

In this section we elaborate the rationale and design details of our fitness function for the language fuzzing.

A. The Machine Learning Model

PCFG Model

Here we present the formal definition of PCFG [24] [25] and the probability calculation.

A PCFG G is a tuple (V, N, S, R, D), where:

- V is a set of terminals, such as string, numbers and identifiers.

```
1  var haystack = "foo";
2  var re text = "ˆfoo";
3  haystack += "x";
4  re text += "(x)";
5  var re = new RegExp(re text);
6  re. test (haystack);
7  RegExp.input = Number();
8  print(RegExp.$1);
```

Fig. 2. This script is from the Mozilla bug report "Regexp object allows arbitrary memory reading " with bug ID 610223 [8]. The root cause [8] is that `RegExp.$1` (Line 8) is a pointer to the first grouped regular expression match while this memory area can be altered by setting a new input (Line 7). The input is not expected to be reset by human programmers. Therefore resetting the input after the **test** is uncommon.

- N is a set of nonterminals N_i, such as statement, for-loop structures in the program.
- $S \in N$ is the start symbol.
- R is a set of rules in the form of $N_i \rightarrow \beta_m$, where $\beta_m = M_1 M_2 \ldots M_n$ is a sequence of terminals and nonterminals.
- D is a function that assigns probabilities to rules, so that

$$\forall i \sum_m D(N_i \rightarrow \beta_m) = 1 \tag{1}$$

We use the PCFG model to calculate the probability of a script directly from the AST tree t using the rule in Equation 2.

$$P(t) \quad = \quad D(N_t \rightarrow \beta) \prod_1^k P(t_{M_i}) \tag{2}$$

where N_t is the root node of t, $\beta = M_1 M_2 \ldots M_k$, and t_{M_i} denotes the N_t's subtrees whose roots are M_i ($1 \leq i \leq k$).

Markov-Chain PCFG Model

The PCFG models the dependencies between N_i and β_m for a rule $N_i \rightarrow \beta_m$. However, dependencies internal a subtree should not be missed.

For example, a script's AST tree is shown in Figure 4 and the grammar is defined as follows(we omit further definitions of different kinds of *stmt* and *param*):

$$\begin{aligned} fun\_def &:= param\_list\ stmt\_list \\ stmt\_list &:= stmt \mid stmt\ stmt\_list \\ param\_list &:= \varepsilon \mid param \mid param\ param\_list \\ stmt &:= assign \mid if \mid while \mid fun\_def \mid \ \ldots \end{aligned}$$

So we may have the rule $stmt\_list \rightarrow fun\_def\ stmt\_list$ and $stmt\_list \rightarrow assign\ stmt\_list$ and so on. The fun_def is `function X()` and the $assign$ is `X.prototype.getName =....` The rules fail to model the dependency between `function X()` and `X.prototype.getName =...` as they don't appear in one rule. Therefore we need to modify the PCFG model.

We choose to use the Markov-Chain model for *stmt_list* and other recursive definitions directly. For a subtree t in the AST and N_t is recursively defined, $N_t \rightarrow \beta_k$ and $\beta_k = M_1 M_2 \ldots M_k$. The probability of $N_t \rightarrow \beta_k$ is computed using a new formula:

```
gczeal(2);
function testCallProtoMethod() {
    function X() {
        this.valueOf =
          new testCallProtoMethod
          ( "return this.value" );
    }
    X.prototype.getName =
      function () { return "X"; }
    var a = [new X, new X,
      new getName, new new this, new Y];
}
testCallProtoMethod();
```

Fig. 3. Test case of bug #746103 of SpiderMonkey.

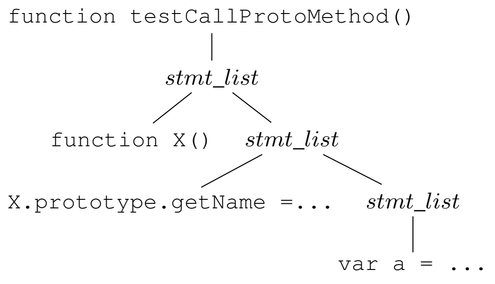

Fig. 4. The AST for the test case in bug #746103, where some nodes are not expanded for brevity.

$$D(N_t) P(M_1 | N_t) \prod_2^k P(M_i | N_t M_1 \ldots M_{i-1}) \tag{3}$$

This formula uses the Markov model to compute the probability for a sequence of statements.

For example,

$$\begin{aligned} & P(\texttt{testCallProtoMethod}) = P(fund\_def) P(\varepsilon | \\ & \quad fun\_def) P(stmt\_list | fun\_def) P(stmt\_list1) \\ & P(stmt\_list1) = P(fun\_def_X) P(assign | fun\_def) \\ & \quad P(var | fun\_def\ assign) \\ & \ldots \ldots \end{aligned}$$

The full conditional probability $P(M_i | N_t, M_1 \ldots M_{i-1})$ is expensive to compute. So we have to compute the approximate probability. If we assume that the probability of M_i is only determined by its proceeding symbol M_{i-1}, then the probability $P(M_1 | N_t) \prod_2^k P(M_i | N_t, M_1 \ \ldots M_{i-1})$ in Equation 3 can be simplified to $P(M_1 | N_t) \prod_2^k P(M_i | N_t, M_{i-1})$. This is a tradeoff between precision and efficiency. The Markov model can model interesting properties of program by assuming that current statement depends on previous K elements [26]. If K is bigger than 1, then the precision is improved while it consumes more memory and computation time.

B. Naturalness Heuristic for Fuzzing

This subsection introduces the techniques of building the fitness function for the fuzzing based on the Markov-Chain PCFG Model. The PCFG and Markov-Chain PCFG model will give lower probability values for longer scripts and this is not desired for our fitness functions.

We first reduce the impact of program length to $P(t)$ by a normalization factor $Z_{fix}(t)$=$(\frac{1}{|N|})^{l_t}$, where t is the AST for the program and l_t is the number of nonterminals contained in t and $|N|$ is the number of available nonterminals in grammar rules. The $Z_{fix}(t)$ is a probability of a sequence of nonterminals in the t under uniform distribution. As the length of the script increases, both $Z_{fix}(t)$ and $P(t)$ decrease at the same time. Therefore $P'(t)$ will be less sensitive to the lengths of scripts.

With $Z_{fix}(t)$, Equation 2 is modified to:

$$P'(t) \quad = \quad Z_{fix}(t)D(N_t \rightarrow \beta_k)\prod_{1}^{k} P(t_{M_i}) \tag{4}$$

Now $P'(t)$ is not a probability as they do not sum to 1. Instead, it measures how uncommon a script is. That is, a script with higher $P'(t)$ suggests it shares more structural features with normal' scripts. If $P'(t)$ is low, then it is more likely to be contain unnatural sequences of grammar elements.

The Markov-Chain PCFG model contains the production $\prod_{2}^{k} P(M_i|N_t, M_1 \ldots M_{k-1})$, and Z_{fix} reduces the value of every conditional probability equally. However not every conditional probability is equally important. We find that many bug-triggering scripts only contain one or two uncommon subtrees among many other subtrees. So we need to amplify the factors of those uncommon subtrees.

We normalize the probability of each subtree recursively instead of normalizing the whole probability by Z_{fix}. Then we use the geometric mean for the whole probability.

$$P^h(t) \quad = \quad \phi\{(1/|N|)^l D(N_t \rightarrow \beta_k)\prod_{1}^{k} P^h(t_{M_i})\} \tag{5}$$

The ϕ function in Equation 5 is adopted to increase the significance those uncommon structures which usually have low probabilities. We chose the square function as ϕ in our current implementation. The design principle is that variance of x^2 is larger than that of x.

$$H(t) = -\log(P^h(t)) \tag{6}$$

The function $H(t)$ is the fitness function of our approach. Scripts with lower values are considered better scripts for the "bug-triggering" object. The scripts with higher scores are considered to be more likely to trigger bugs.

C. Parameter Training

The values of $D(N_i)$, $P(M_1|N_i)$ and $P(M_i|N_t, M_{i-1})$ in Equation 3 can be learned from a large, training corpus of scripts. The $D(N_i)$ is calculated as:

$$D(N_i) = \frac{\#N_i}{\#N}$$

where $\#(N_i)$ is the number of occurrence of N_i and $\#N$ the number occurrence of rules in the training corpus.

Similarly, the conditional probability $P(M_i|N_t, M_{i-1})$ can be approximated by:

$$P(M_i|N_t, M_{i-1}) = \frac{\#M_i}{\#M_{i-1}M_i}$$

where $\#M_i$ and $\#M_{i-1}$ are statisticed for all rules $N_t \rightarrow \beta_m$ in the training corpus.

Our model tries to catch the uncommonness of a script. So if the sequence $M_{i-1}M_i$ does not occur in the training corpus then we consider that this sequence is more likely to be odd and we set probability $P(M_i|M_{i-1})$ to the smallest float value allowed by CPU instead of zero.

V. Evaluation

In this part we evaluate our method. We compare our fitness function with pure PCFG-based method, the fitness function in IFuzzer which is the state-of-art GP-based language fuzzing. We evaluate the accuracy and precision performance of these functions in selecting the candidates in fuzzing.

A. Evaluation Dataset Setup

In our experiment, 150,000 JavaScript scripts [27](100,000 for parameter training) and 900 bug-triggering JavaScript scripts(all for testing) from Mozilla's Bugzilla are used to evaluate the three fitness functions. We select all the bug-triggering scripts from confirmed crashing reports for the SpiderMonkey ranging from year 2000 to now (some crash reports provide trace only and we just omit them). These scripts are extracted from the report texts or in the attachments. We called the set of bug-triggering scripts from Bugzilla as the bug group, and the 50,000 scripts from [27] as the normal group.

The scripts in normal group are mostly written for real world projects, so it is unlikely that the scripts will trigger bugs in interpreters which have been tested on them before release. Therefore we assume the scripts in the normal group will not trigger bugs.

Our bug-triggering scripts from the bug group are version independent of SpiderMonkey. The reason is that it is difficult to determine if a script can trigger bugs in without specific version of JavaScript interpreter. It is possible that a script does not trigger bugs in SpiderMonkey 32, but it may trigger bugs in other versions of SpiderMonkey or other interpreters like V8 and Webkit.

Experiment settings: We use Acorn [28] as the parser for the JavaScript scripts considered. A prototype of our NHF function has been implemented using Python. In addition, the experiment was performed on a computer with Intel i7-6700 CPU at 3.40GHz and 8G memory.

B. Evaluation Measures

We use the ROC(Receiver Operating Characteristic) as the evaluation method. One reason is that ROC is well accepted by machine learning community. The other reason is that fuzzing is not deterministic and the number of bugs found may subject to large deviations. The fitness function gives scores to scripts and we can classify scripts with scores higher than θ_s as *bug-triggering*. So we can use the ROC curve to evaluate the performance of fitness functions in ranking scripts. For a given threshold θ_s, if the number of test cases whose fitness values are higher than θ_s is B and the number of scripts in both groups whose fitness are higher than θ_s is N. The precision is defines as B/N and the recall is defined as $B/(T-B)$, where T is the number of scripts in the bug group. It should be noted that the threshold is not used in the fuzzing because we just select top K elements in the GP iterations.

C. Evaluation Results

The frequency histograms of the fitness values calculated by IFuzzer, PCFG and NHF for the JavaScript scripts are illustrated in Figure 5, 6, 7 respectively. The horizontal axes of these Figures denote the normalized fitness values. (i.e., the fitness value of a script divided by a manually selected value so that the graph can be displayed clearly and the shape of distribution be reserved)[1] In the histograms, the scripts with

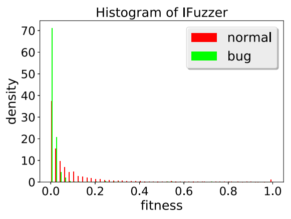

Fig. 5. IFuzzer

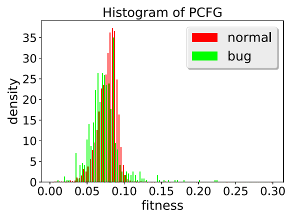

Fig. 6. PCFG

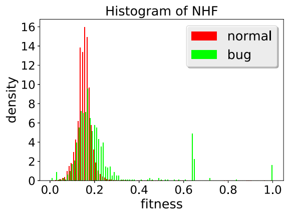

Fig. 7. NHF

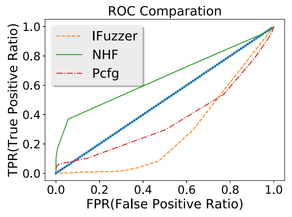

Fig. 8. ROC curves of PCFG, IFuzzer and NHF

Fig. 9. The distribution of bug group moves towards right in th histogram for NHF, the two groups have similar distribution and peaks for PCFG, and the two groups have similar distribution and peaks for IFuzzer.

higher fitness values are expected to be in the “bug-group”. The distance of two groups should be large in ideal. However, in reality the normal and bug group overlap. It can be noticed that in the NHF's histogram, the normal group and bug group has a larger divergence.

[1]The reason to normalize the values of JavaScript scripts under analysis is to enable the display of experiment data in the figure. Similar normalization can also be found in Figures 6 and 7.

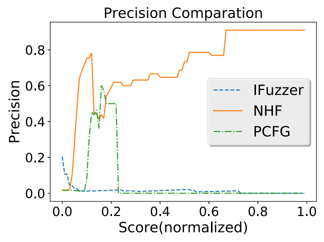

Fig. 10. The Ranking Precisions.

Figure 8 shows the ROC curves for IFuzzer, NHF and PCFG. It shows that the ROC curves for IFuzzer's fitness function and the PCFG method both are below the diagonal line indicating that they don't perform better than blind-guessing in terms of predicating whether a script is bug-triggering.

In contrast, the ROC curve of our method is above the diagonal line and this means that it is better than PCFG and IFuzzer. As we use this fitness function for fuzzing, we do not need to choose a threshold. We only select top K scripts for the next generation of population during the iterations of GP. Figure 10 shows the precision rates of the three fitness functions when used as a classifier. We show the precision (the Y-axis) for different thresholds(X-axis). This demonstrates that accuracy of top K scripts. The precision for IFuzzer is low and not improved as the threshold increases. The precision for PCFG is high when the threshold is around 0.2 and starts to decrease significantly when the threshold is set higher than 0.2. This is because some scripts in normal group are long enough to have higher values than scripts in the bug group. Our method has a robust performance for large enough threshold and the precision is over 80% if the threshold is larger than 0.6.

We also notice that IFuzzer tends to calculate higher fitness values for complex, long scripts in both the normal and bug-trigger groups. To quantify the correlation between the fitness value and the length of a script, we calculate the correlation coefficients as follows:

$$r(S,L)=\frac{cov(S,L)}{\sqrt{var(S)var(L)}}$$

where S is the fitness value, L is the script length, $cov(S,L)$ is the covariance of S and L, and $var(X)$ and $var(L)$ are the variances of X and L, respectively. In particular, $cov(S,L)$ is calculated as follows:

$$cov(S,L)=\frac{\sum_{i=1}^{n}(S_i-\bar{S})(L_i-\bar{L})}{n-1}$$

Table I summarizes the correlation coefficients of script fitness values and lengths for different groups of datasets. The correlation coefficient for IFuzzer is high. It should be noticed that the correlation coefficient is larger in the normal group than that in the bug group for IFuzzer. The reason is that

Name	Data Group	Correlation
IFuzzer	normal	0.7904
	bug	0.1583
PCFG	normal	0.1051
	bug	0.1132
NHF	normal	0.0196
	bug	-0.0103

TABLE I
CORRELATION COEFFICIENTS BETWEEN FITNESS VALUES AND SCRIPT LENGTH CALCULATED BY THREE FITNESS FUNCTIONS

the scripts in the bug group usually contain relatively simple structures and as a result most of their fitness values are small. The PCFG method has smaller correlation coefficients. Our method further reduces the correlation between fitness values and script lengths. This might suggest that, with our method, the fitness value of a script is almost independent of its lengths.

VI. CONCLUSION

We have presented a novel fitness function that integrates Markov-chain and PCFG models for the concept of "bug-triggering" for language fuzzing. The Markov-chain models the dependencies between adjacent elements in the list for a recursive rule. We use a normalization method to reduce the influence of the script lengths so that the long common scripts will not be considered as uncommon scripts. We evaluate the effectiveness of our method using a large corpus of JavaScript application codes from Github and Firefox's Bugzilla. Experiment results show that our method's accuracy is better than that of IFuzzer and PCFG-based probability. Our model is language independent and can also be applied to fuzzing of interpreters for other languages. We will develop a more complete fuzzing tool in future work.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant No. 61471344.

REFERENCES

[1] "The official mirror of the v8 git repository," https://chromium.googlesource.com/v8/v8.git.

[2] "Spidermonkey (javascript-c) engine," https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[3] "Open source web browser engine," https://webkit.org/blog/189/announcing-squirrelfish/.

[4] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb 2013.

[5] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[6] M. Alshraideh, B. A. Mahafzah, and S. Al-Sharaeh, "A multiple-population genetic algorithm for branch coverage test data generation," *Software Quality Journal*, vol. 19, no. 3, pp. 489–513, sep 2011.

[7] S. Veggalam, , S. Rawat, , I. Haller, , and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *ESORICS 2016: 21st European Symposium on Research in Computer Security*. Heraklion, Greece: Springer International Publishing, 2016, pp. 581–601.

[8] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21th USENIX Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 445–458.

[9] J. Ruderman, "Introducing jsfunfuzz," http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[11] J. D. Demott, D. Richard, J. Enbody, D. William, and F. Punch, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," *Blackhat and Defcon*, 2007.

[12] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8636 LNCS, 2014, pp. 138–152.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," *NDSS'17*, no. March, 2017.

[14] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 761–774.

[15] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 49–65.

[16] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 708–719.

[17] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for finding api usage errors," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 151–162.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[19] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 428–439.

[20] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine Learning for Input Fuzzing," Tech. Rep., 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/learnfuzz-machine-learning-for-input-fuzzing/

[21] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 579–594.

[22] J. Patra, M. Pradel, J. Patra, and M. Pradel, "Learning to Fuzz : Application-Independent Fuzz Testing with Probabilistic , Generative Models of Input Data Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic , Generative Models of Input Data," Tech. Rep., 2016. [Online]. Available: http://mp.binaervarianz.de/TreeFuzz{_}TR{_}Nov2016.pdf

[23] "Javascript reference in MDN web docs," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/test.

[24] C. D. Manning and H. Schtze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[25] M. Johnson, "PCFG models of linguistic tree representations," *Comput. Linguist.*, vol. 24, no. 4, pp. 613–632, dec 1998.

[26] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, nov 2014, pp. 269–280.

[27] "Learning from "big code"," http://learnbigcode.github.io/datasets/.

[28] "A small, fast, javascript-based javascript parser," https://github.com/ternjs/acorn.