A Feature-Oriented Corpus for Understanding, Evaluating and Improving Fuzz Testing

Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao
xiaogangzhu@swin.edu.au
Swinburne University of Technology
Melbourne, VIC

Sheng Wen, Yang Xiang
swen,yxiang@swin.edu.au
Swinburne University of Technology
Melbourne, VIC

Seyit Camtepe
Seyit.Camtepe@data61.csiro.au
DATA61 | CSIRO
Sydney, NSW

Jingling Xue
jingling@cse.unsw.edu.au
The University of New South Wales
Sydney, NSW

ABSTRACT

Fuzzing is a promising technique for detecting security vulnerabilities. Newly developed fuzzers are typically evaluated in terms of the number of bugs found on vulnerable programs/binaries. However, existing corpora usually do not capture the features that prevent fuzzers from finding bugs, leading to ambiguous conclusions on the pros and cons of the fuzzers evaluated. In this paper, we propose to address the above problem by generating corpora based on search-hampering features. As a proof-of-concept, we designed FEData, a prototype corpus that currently focuses on three search-hampering features to generate vulnerable programs for fuzz testing. Unlike existing corpora that can only answer "how", FEData can also further answer "why" by exposing (or understanding) the reasons for the identified weaknesses in a fuzzer. The "why" information serves as the key to the improvement of fuzzers. Based on the "why" information, our FEData programs enabled us to identify the weakness of AFLFast, called *cycle explosion*, behind. We further developed an improved version of AFLFast, called `AFLFast+`, which has overcome the cycle explosion problem. AFLFast+ retains the efficiency of AFLFast in path search while maintaining or even surpassing the bug-finding capability of AFL for the corpus evaluated.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Fuzzing, Feature-Oriented Corpus, Evaluation

ACM Reference Format:
Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. 2019. A Feature-Oriented Corpus for Understanding, Evaluating and Improving Fuzz Testing. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3321705.3329845

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329845

1 INTRODUCTION

Fuzzing is an automatic software testing technique that typically provides random data as inputs to programs and then monitors them for exceptions such as crashes. Fuzzing can capture bugs[1] because the exceptions are usually the indicators of bugs in the program context. In 1990, B.P. Miller *et al.* [17] developed the first fuzzing algorithm (fuzzer for short). Since then, fuzzing has become one of the major tools for detecting bugs.

Though many fuzzers have been developed, their appropriate evaluation is still a big challenge due to the lack of supportive corpora. To validate a newly developed fuzzer, a corpus needs to contain the contexts of bugs such as specific search-hampering features for fuzzing. Those contexts can help validate the advancement of a fuzzer resolving the specific challenges. However, it is not feasible for fuzzing so far. In current stage, the usual way to evaluate fuzzers is to run them on real-world program corpora, and judge their performance by counting the number of unique crashes after a period of running time (normally 24 hours) [7–9, 13, 19, 22, 26]. As disclosed by G. Klees *et al.* [14], this often leads to ambiguous or even wrong conclusions on the fuzzers, since it is almost impossible to provide supportive contexts of bugs by inspecting a large number of programs in corpora. A simple example is that, Driller [25] detects more bugs than AFL [2] by utilizing concolic execution [23] to resolve magic values. But the evaluation results may not support the advancement of Driller when the corpora do not have enough bugs 'protected' by magic values. In fact, even though we reckon the evaluation results are convincing, we still cannot pinpoint the reason that leads to fuzzers' advancements or weaknesses without contextual details of bugs in corpora. This hampers or even disables our attempts on improving existing fuzzers.

In this paper, we propose generating corpora based on search-hampering features to solve the above challenge. To this end, a framework is developed and used to synthesize evaluation corpora in an automatic manner. The vulnerable programs are made up by

[1] Vulnerability is different from bug. Bug brings a program into an unintended state. When a bug can be exploited by an attacker, the bug becomes a vulnerability [18]. In the following of this paper, we will use the word "bug" instead of "vulnerability" for brevity, but what we mean is "vulnerability".

function-level structures from GitHub[2] code to maintain as much as possible the style of real-world programs. The framework ensures the generated programs are able to be compiled, and inserts the contexts of bugs to the programs when necessary. Holding contexts of bugs, the generated corpora can not only make the conclusions of a fuzzer's advancements solid, but also expose the clues for improving a fuzzer. To demonstrate the effectiveness of our idea, we develop a proof-of-concept corpus, namely `FEData`, based on three typical search-hampering features *i.e.*, the number of magic values, the number of execution paths, and the number of checksums.

As a case study, we run AFL and AFLFast [7] on programs from FEData to validate its utility in fuzzing evaluation. The results of our experiments confirm the conclusion made by AFLFast that it finds execution paths faster than AFL. The results also indicate that AFLFast detects fewer bugs than AFL in some specific programs from FEData. G. Klees *et al.* [14] found similar phenomenon that AFLFast detected fewer bugs when they ran these two fuzzers more than 24 hours, but they did not explain the reason for it. With contexts of bugs on those programs, we find that AFLFast is prone to fall into the `cycle explosion` state. As AFLFast can no longer produce new inputs in this state, it will find fewer bugs than AFL. Accordingly, we improve AFLFast and develop `AFLFast+` by setting a lower bound to the number of inputs produced by AFLFast, which prevents fuzzing from dropping into the cycle explosion state.

We summarize the contributions as follows:

- We propose generating corpora for fuzzing evaluation based on search-hampering features. The generation runs in an automatic manner, and the generated corpora are not only used for understanding and evaluating fuzzing but also helping improve the fuzzers based on the contexts of bugs.
- We carry out a case study on AFLFast via a proof-of-concept corpus, called FEData. The corpus is generated based on three typical search-hampering features. The experimental results validates the utility of the feature-oriented corpus by showing its accurate conclusions in fuzzing evaluation.
- We develop AFLFast+ according to AFLFast's weakness that we expose in the case study. The experiments show that AFLFast+ can detect more bugs than AFLFast when other factors stay the same.

2 OVERVIEW OF THE IDEA

The idea of generating feature-oriented corpora will help solve the challenge of fuzzing evaluation. To make the idea clearer, we take two fuzzers to exemplify the necessity of holding contexts of bugs for fuzzing evaluation. We take the evaluation of Driller [25] against AFL [2] as an example. Driller claims to beat AFL because it can resolve magic values more efficiently. In fact, Driller is an updated version of AFL since it takes AFL as the core and adopts concolic execution to resolve magic values for larger coverage. However, the claim of Driller may not be supported in fuzzing evaluation sometimes. As shown in Fig.1(A), because there are no magic values on the execution path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow G$, the concolic execution in Driller makes no difference from AFL to trigger the bug. Even worse, the concolic execution in Driller takes more time to resolve the magic value $0x126$ in an execution path where there is no bug. Considering the randomness nature of fuzzing, it is highly possible for Driller to have a worse performance than AFL. However, Driller can outperform AFL on the programs similar to Fig.1(B), where there are lots of bugs hidden behind magic value challenges. Therefore, when state-of-art corpora do not provide contexts of bugs for evaluation, the results may lead to ambiguous or even wrong conclusions on fuzzers.

[2]https://github.com

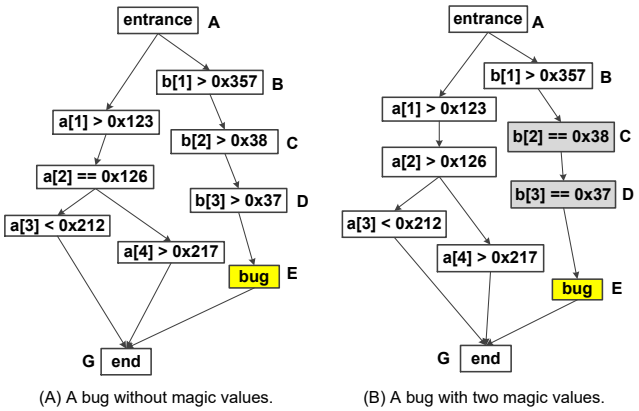

Figure 1: Two program examples that have different numbers of magic values to protect bugs. It may lead to a wrong conclusion if fuzzers such as Driller and AFL are evaluated on program (A). Driller outperforms AFL only on program (B) since bug is hidden behind two magic value challenges.

It is necessary to implement contexts of bugs such as search-hampering features in fuzzing evaluation corpora. As shown in Fig.2, the corpus of vulnerable programs is one of the major components in fuzzing evaluation procedure. Fuzzers follow a policy such as how to generate seeds, and run on the evaluation corpus. The efficiency of fuzzing is mainly determined by the capability of the fuzzer solving contextual challenges in programs, such as magic values, checksums, path search, *etc.* We can therefore measure the performance of fuzzers via a list of metrics such as the number of magic values resolved in the execution. However, to date, the usual way to judge the performance of a fuzzer is to count the number of bugs exposed within a fixed period of time. Because the number of exposed bugs may not be positively correlated to the challenges, it cannot always indicate the efficiency of a fuzzer. In addition, since the number of exposed bugs cannot be linked to the metrics without contexts of bugs in programs, the usual way becomes too superficial to diagnose the drawbacks of a fuzzer. This explains our technical motivation as well as the real need to develop fuzzing evaluation corpora that implement contexts of bugs such as search-hampering features in vulnerable programs.

We create a prototype framework to automatically generate corpora for fuzzing evaluation. The framework uses real-world program structures from GitHub to maintain the synthetic coding style. The contexts of bugs are then inserted into the structures, followed by extra code that ensures the generated programs are able to be compiled. Different from existing synthetic corpora, the design of our evaluation corpora is determined by the type(s) of

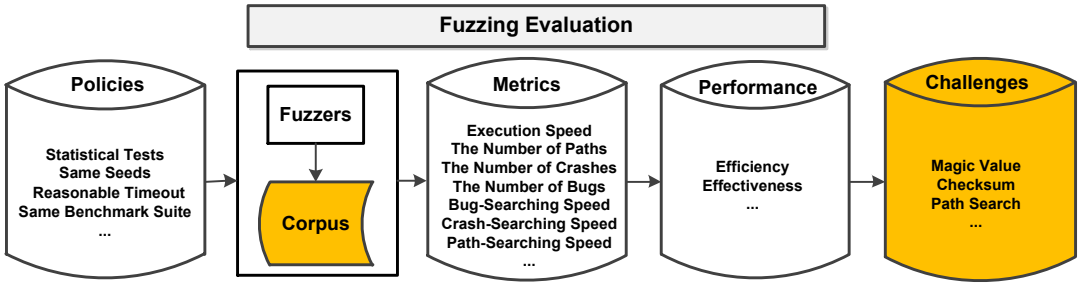

Figure 2: The procedure for evaluating fuzzers. Policies limit the way to run fuzz testing. The performance of a fuzzer can be evaluated based on the metrics, which is mainly determined by the capability of solving challenges.

search-hampering features that the fuzzer focuses on. In other words, the design is more 'feature-oriented' for fuzzing evaluation.

3 SEARCH-HAMPERING FEATURES

Search-hampering feature is the major factor that affects the fuzzing performance and their evaluations according to the above analysis. In fact, there are many search-hampering features. In this section, we only select and focus on three typical ones: 1) the number of execution paths, 2) the number of magic values, and 3) the number of checksums. These three features are important due to the challenges that fuzzing attempts to solve. As a program consists of three fundamental control structures, which are sequence structure, decision structure, and loop structure, fuzzers have to find solutions for the following challenges, 1) checksums, 2) magic values, and 3) execution path search, in order to reach larger code coverage. To date, many fuzzers [7, 8, 12, 15, 19, 21, 25] have been proposed to improve the performance of solving these three challenges.

According to the three search-hampering features, we design a prototype corpus, called FEData. FEData generates simple programs and each of the programs has only one bug.

3.1 The Number of Execution Paths

The number of execution paths affects the efficiency of fuzzing. Fuzzing spends much time to locate the bug hidden in a large number of execution paths. Fig.3 shows different numbers of execution paths, where Fig.3(A) has four execution paths, and 3(B) has 200 execution paths. Fuzzing may spend a few seconds to find the bug hidden in 3(A), but it will cost fuzzing much more time to locate the bug hidden in 3(B).

3.2 The Number of Magic Values

A magic value is a constant in the context of programs. In fact, the nature of fuzzing is to generate inputs and find bugs automatically. To detect bugs, fuzzers have to pass through checks and find the bug path. Therefore, it will cost fuzzing more time to find bugs if the bugs are protected by magic values. We recall the example in Fig.1(B) to explain the magic values. In this example, the constants $0x38$ and $0x37$ are magic values as they are sitting in one side of the '==' conditions in the *if* statements. Moreover, because fuzzers regard every path as a potential bug path, they will attempt to resolve magic values in every path. Therefore, magic values are important to the efficiency of fuzzing, but it is not always true

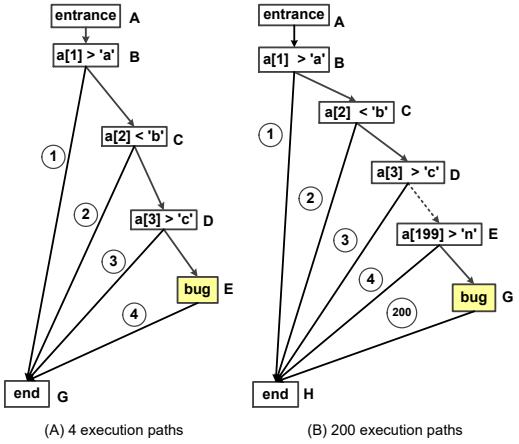

Figure 3: Different numbers of execution paths. It will cost fuzzing more time to locate the bug in (B) than in (A).

```
1  int main(int argc, char ** argv){
2    char inp1[6], inp2[6];
3    fgets(inp1, 6, stdin);
4    fgets(inp2, 6, stdin);
5    if(!strcmp(inp1, "MAGIC")) //magic value
6    {
7      printf("Not a bug.");
8    }
9    if(!strcmp(inp2,"BYTE")) //magic value
10   {
11     bug();
12   }
13   return 0;
14 }
```

Listing 1: Magic values. Magic value 'BYTE' protects the bug while 'MAGIC' does not.

for a fuzzer to expose more bugs when it resolves magic values more efficiently. As an example, Listing 1 shows two magic values, wherein the magic value 'MAGIC' does not protect the bug while the magic value 'BYTE' does.

```
bool func_checksum(int a[7]){
    if (length(a)!=7) return False;
    if (sum(a)%8==3){
        return True;
    }
    else{
        return False;
    }
}
```

Listing 2: A simple example of checksum function. Two steps are taken to ensure that the variable a satisfies the specific situations.

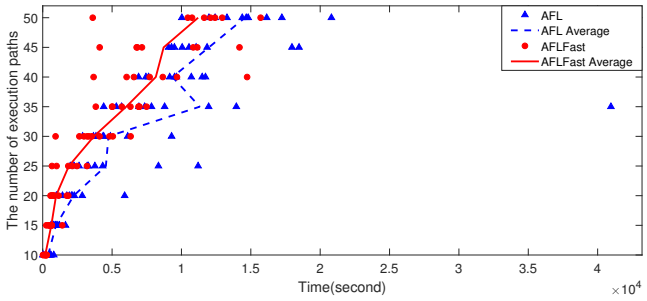

Figure 4: AFL and AFLFast are run on 90 binaries. One dot or triangle indicates an identified bug.

3.3 The Number of Checksums

A checksum is designed to detect errors in a block of digital data. The effect of checksum is similar to the magic value but resolving a checksum is more complex because checksum needs complicated calculations. A checksum function is a function that helps process the procedure of calculations. This function can be utilized in a control statement condition, such as $if(func\_checksum(a))$. In this scenario, a checksum can be regarded as a more complicated magic value. For example, Listing 2 shows a simple checksum function. In this function, two *if* statements are introduced to check if specific conditions have been satisfied: 1) '$length(a) == 7$' and 2) '$sum(a)\%8 == 3$'. We can see from this example that, to resolve a checksum is more complicated than to resolve a magic value.

4 EVALUATION

Experiments are run on AMD Opteron 6320, and each fuzzer is run on one core. We typically run two fuzzers, AFL and AFLFast [7]. AFL is a greybox fuzzing regarded as the baseline by many other fuzzers. It initially sets some seeds and produces many inputs mutated from these seeds. Then, those inputs will be fed into the testing program, and new seeds will be selected from the inputs based on the results of the execution. More inputs will be generated from the new seeds and those inputs will also be fed into the testing program. This procedure will be processed repeatedly. A cycle is done when all the seeds are chosen and mutated to produce inputs, and all the generated inputs have been fed into the testing program. Then, a new cycle begins. Many fuzzers, including AFLFast, develop their own algorithms based on AFL [7, 12, 22, 25].

The experiment is designed to assess the ability of the two fuzzers to search specific execution paths. In this experiment, 90 binaries, which contain different numbers of execution paths and have no specific checks (*i.e.,* magic value or checksum) in the bug path, are used to evaluate AFL and AFLFast. We stop fuzzing when it has found the inserted bug or it has run for 12 hours.

As for the policies used for evaluation, we set the same seed, use statistical results, and choose the same programs for both AFL and AFLFast. Specifically, both of the two fuzzers use Hello World as the original seed. Besides, we use statistical results to evaluate fuzzers, *i.e.,* the average time of finding a bug. Moreover, these two fuzzers are evaluated on the same programs and set the same timeout. We utilize four metrics, including the execution speed, the number of bugs found, the number of identified execution paths, and the number of cycles that have been executed.

During this experiment, we find that AFLFast may run too many cycles, more than what it needs. This is because AFLFast no longer produces inputs and runs through a cycle very fast. We call this phenomenon `cycle explosion`. The cycle explosion is a state that a fuzzer can no longer produce new inputs and will run a large number of cycles. The results are shown in Fig.4.

In Fig.4, either a dot or a triangle denotes a bug found by fuzzing. The result indicates that AFLFast indeed finds bugs faster than AFL. Fig.4 also shows that AFLFast searches execution paths faster than AFL in each kind of programs. When the number of execution paths grows, it costs both AFL and AFLFast more time to trigger bugs. This trend supports that the number of execution paths affects the time for fuzzing to find bugs.

However, AFLFast finds fewer bugs than AFL, especially when the number of execution paths grows larger. G. Klees *et al.* also give a similar conclusion that AFLFast may perform worse than AFL when evaluating longer than 24 hours. We explore the reason for this conclusion and find an interesting phenomenon. The reasons that AFLFast cannot find some bugs are different from AFL. All the four bugs that AFL cannot find is due to the limited time. However, all the 13 bugs that AFLFast cannot find are because of cycle explosion. Based on Fig.4, the conclusion is that AFLFast finds bugs faster than AFL, but it has a chance to get trapped in the cycle explosion, especially when the bugs are protected in a deep path.

An Interesting Case. One binary is chosen from the 90 binaries and the result is shown in Fig.5. It shows that AFLFast stops finding new execution paths after the 6360th second. After the 24000th second, the execution speed of AFLFast drops to 0/s. Meanwhile, the number of cycles done by fuzzing grows to a large number. That's the state when AFLFast is trapped in cycle explosion. On the other hand, AFL executes binaries at the speed about 50/s.

5 IMPROVEMENT OF AFLFAST (AFLFAST+)

It is impressive that the cycle explosion prevents AFLFast from triggering bugs. Therefore, we dig deeper into the reason for it and present a solution to improve AFLFast. AFL assigns almost constant energy (*i.e.,* the number of inputs generated from seeds) to each seed. Therefore, AFL does not cause cycle explosion and even can find bugs in one cycle. AFLFast improves AFL by assigning different number of inputs to a seed. Different mutation strategies, including FAST mode, LINEAR mode, and QUAD mode, are utilized by AFLFast.

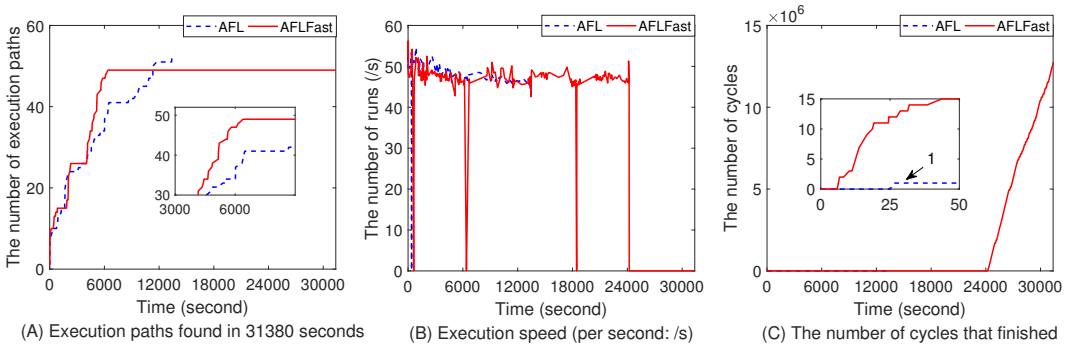

Figure 5: An interesting case. In the first 3600 seconds, AFL and AFLFast find execution paths at almost the same speed. From the 3600th second to the 6360th second, AFLFast finds execution paths faster than AFL. The interesting part is that AFLFast cannot find more execution paths since the 6360th second while AFL finds more execution paths until it crashes the program.

The FAST mode assigns energy to state i as

$$p_f(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right) \quad (1)$$

Where $\alpha(i)$ is the number of inputs generated by AFL, $s(i)$ is the number of times that seed d_i is chosen from the seed queue, $f(i)$ is the number of generated inputs that have exercised state i. The constant M provides an upper bound on the number of inputs that are generated per fuzzing iteration.

The LINEAR mode increases the energy of state i linearly with respect to $s(i)$, which is

$$p_l(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right) \quad (2)$$

The QUAD mode is based on the LINEAR mode, and the energy of fuzzing is computed as

$$p_d(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right) \quad (3)$$

All these three power schedules assign energy to state i inversely proportional to $f(i)$, which is the reason for cycle explosion. When $f(i)$ grows to a large value, the energy is very low. To improve AFLFast, one way is to set a lower bound to its energy strategies. L is a lower bound set for AFLFast. Therefore, AFLFast can be improved as

$$q(i) = \max\left(p(i), L\right) \quad (4)$$

where $p(i)$ stands for $p_f(i), p_l(i)$, or $p_d(i)$, and $q(i)$ is the improvement of $p(i)$ respectively.

Evaluation On the 90 Binaries. We evaluate our AFLFast+ on the 90 binaries from the above experiment. Fig.6 shows that the efficiency of AFLFast and AFLFast+ is almost the same. However, AFLFast+ finds 88 bugs while AFLFast finds only 77 bugs. Meanwhile, the reason of the two bugs not being found by AFLFast+ is due to the limited time, and the cycle explosion does not appear. Therefore, AFLFast+ keeps the efficiency but can detect more bugs than AFLFast.

6 RELATED WORK

The performance of a fuzzer is mainly determined by its capability of handling search-hampering features in the contexts of bugs.

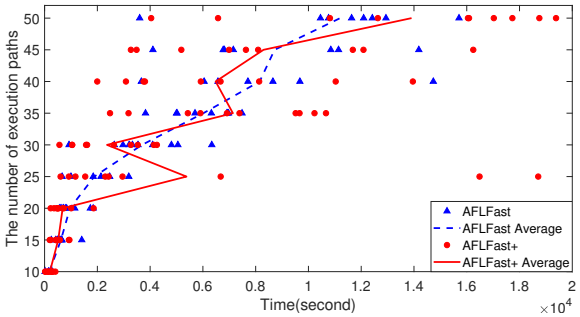

Figure 6: The results of AFLFast and AFLFast+ which are tested on 90 binaries. One dot or triangle is a bug found.

As search-hampering features are important to the fuzzing performance, most fuzzers [7, 9, 12, 15, 19–21, 25, 26] are tailored to resolve one/several features to increase the coverage of code.

So far, researchers have already made some efforts to address the above evaluation challenge. For example, some researchers prefer to create corpora based on real-world programs, typically from student code [24], existing bug report databases [16], or by creating a public bug registry [3, 11]. Despite these proposals provide corpora contextual details of bugs, they have remained static and relatively small for fuzzing evaluation. There are also some tries on creating independently defined public benchmark suites, *e.g.* DaCapo [6] and SPEC [5]. Among these attempts, even though they collect a large volume of real-world vulnerable programs, it is still painstaking to manually triage the crashes and filter the bugs in those programs. Before a strong community effort has been applied to these suites, it is believed that they are not sufficient to support fuzzing evaluation [14].

An alternative way to create fuzzing corpora is to synthesize vulnerable programs. Typical examples include early corpora for buffer overflow detection [27, 28], LAVA and next version LAVA-M [10], DARPA CGC corpus [1], as well as corpus drawn from NIST SAMATE project [4]. The synthetic corpora ensure the existence of bugs by sacrificing their reflection on real-world ones. However, to the best of our knowledge, none of the above synthetic corpora

have implemented contexts of bugs pertaining to search-hampering features. Therefore, synthetic corpora also cannot support fuzzing evaluation according to aforementioned analysis.

7 DISCUSSION: THREAT TO VALIDITY

Although FEData works well on AFL and AFLFast, it can be further improved. Issues such as the ways to evaluate FEData and improve FEData can be further discussed.

Firstly, we only run AFL and AFLFast on FEData because other fuzzers are unavailable or cannot be appropriately run. In order to further assess FEData, different fuzzers should be run on FEData. In order to assess the feature of magic value in FEData, a better way is to use fuzzers trying to help resolve magic values, such as Driller and T-Fuzz. Similarly, it is better to assess the feature of checksum in FEData by fuzzers focusing on checksums. We will continue our investigation to run more fuzzers on FEData.

Secondly, to customize search-hampering features in the contexts of bugs, FEData sacrifices some realness of the generated programs. To make FEData more similar to real-world programs, one way is to add functionality by functional programming. Another way to improve the realness of FEData is to research the contexts behind bugs, which can be added to FEData when a bug is inserted.

8 CONCLUSION

In order to evaluate fuzzing more specifically and effectively, we propose generating corpora based on search-hampering features. Details about the features are described in this paper. Further, we design a prototype corpus, FEData, to show the effectiveness of our idea. AFL and AFLFast are evaluated on FEData. The advancement of AFLFast, which is AFLFast finds execution paths faster than AFL, is supported by FEData. However, the drawback of AFLFast is magnified by some programs in FEData. We find that AFLFast is prone to be trapped in cycle explosion if bugs are embedded deeply or fuzzers hit some specific execution paths in a large quantity. Therefore, AFLFast is improved to AFLFast+ via setting a lower bound to its energy strategies. The results show that AFLFast+ can find more bugs than AFLFast while the efficiency stays the same.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Toby Murray and Dr. Jianhai Chen. Thanks for discussing with us on important issues.

REFERENCES

[1] . 2017. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/.
[2] . 2018. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[3] . 2018. Fuzze Test Suite. https://github.com/google/fuzzer-test-suite.
[4] . 2018. National Institute of Standards and Technology. https://www.nist.gov.
[5] . 2018. Standard Performance Evaluation Corporation. https://www.spec.org/benchmarks.html.
[6] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41(10). ACM, ACM, New York, NY, USA, 169–190.
[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, IEEE, Vienna, Austria, 1032–1043.
[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 711–725.
[9] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, ACM, Dallas, Texas, USA, 2123–2138.
[10] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale automated vulnerability addition. In *Security and Privacy, 2016 IEEE Symposium on*. IEEE, IEEE, SAN JOSE, CA, USA, 110–121.
[11] Jeffrey Foster. 2005. A call for a public bug and tool registry. In *Workshop on the Evaluation of Software Defect Detection Tools*.
[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy*. IEEE, IEEE, San Francisco, CA, USA, 679–696.
[13] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 49–64.
[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, ACM, Toronto, Canada, 2123–2138.
[15] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, ACM, Paderborn, Germany, 627–637.
[16] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.
[17] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.
[18] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, San Diego, CA, USA.
[19] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy*. IEEE, IEEE, San Francisco, CA, USA, 697–710.
[20] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, IEEE, Singapore, Singapore, 543–553. https://doi.org/10.1145/2970276.2970316
[21] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, February 26 - March 1, 2017*. The Internet Society, San Diego, California, USA.
[22] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182.
[23] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 571–572.
[24] Jaime Spacco, David Hovemeyer, and William Pugh. 2005. Bug specimens are important. In *Workshop on the Evaluation of Software Defect Detection Tools*.
[25] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *3rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, Vol. 16. NDSS, The Internet Society, San Diego, California, USA, 1–16.
[26] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy*. IEEE, IEEE, San Jose, CA, USA, 579–594.
[27] John Wilander and Mariam Kamkar. 2003. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention.. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003*, Vol. 3. The Internet Society, San Diego, California, USA, 149–162.
[28] Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Vol. 29(6). ACM, ACM, Newport Beach, CA, USA, 97–106.