PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode

Jie Liang
KLISS, BNRist, School of Software,
Tsinghua University, China
liangjie.mailbox.cn@gmail.com

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University, China
jiangyu198964@126.com

Yuanliang Chen
KLISS, BNRist, School of Software,
Tsinghua University, China
chenyuan17@mails.tsinghua.edu.cn

Mingzhe Wang
KLISS, BNRist, School of Software,
Tsinghua University, China
wmzhere@gmail.com

Chijin Zhou
KLISS, BNRist, School of Software,
Tsinghua University, China
tlock.chijin@gmail.com

Jiaguang Sun
KLISS, BNRist, School of Software,
Tsinghua University, China
sunjg@tsinghua.edu.cn

ABSTRACT

Researchers have proposed many optimizations to improve the efficiency of fuzzing, and most optimized strategies work very well on their targets when running in single mode with instantiating one fuzzer instance. However, in real industrial practice, most fuzzers run in parallel mode with instantiating multiple fuzzer instances, and those optimizations unfortunately fail to maintain the efficiency improvements.

In this paper, we present PAFL, a framework that utilizes efficient guiding information synchronization and task division to extend those existing fuzzing optimizations of single mode to industrial parallel mode. With an additional data structure to store the guiding information, the synchronization ensures the information is shared and updated among different fuzzer instances timely. Then, the task division promotes the diversity of fuzzer instances by splitting the fuzzing task into several sub-tasks based on branch bitmap. We first evaluate PAFL using 12 different real-world programs from Google fuzzer-test-suite. Results show that in parallel mode, two AFL improvers–AFLFast and FairFuzz do not outperform AFL, which is different from the case in single mode. However, when augmented with PAFL, the performance of AFLFast and FairFuzz in parallel mode improves. They cover 8% and 17% more branches, trigger 79% and 52% more unique crashes. For further evaluation on more widely-used software systems from GitHub, optimized fuzzers augmented with PAFL find more real bugs, and 25 of which are security-critical vulnerabilities registered as CVEs in the US National Vulnerability Database.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Software testing, Parallel, Fuzzing

ACM Reference Format:
Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275525

1 INTRODUCTION

Despite the prudent software developing and testing process, vulnerabilities are still common in software [11–13]. To find these vulnerabilities and ensure security, researchers have made a lot of efforts and proposed many techniques in software testing. Mutation-based fuzzing relies on repeatedly feeding the program with modified inputs and monitoring the abnormal behaviours [16]. Compared with other vulnerabilities detecting techniques, better scalability and less manual efforts contribute to its popularity in industry [1–4, 10, 15, 21]. However, as the continuous growth of software complexity, trivial mutation-based fuzzers are hard to ensure the code coverage. For example, they may be blocked by input format checks and ignore vulnerabilities which hide in hard-to-reach areas.

One popular idea to improve mutation-based fuzzing is getting more program information to guide deliberate input modifications. For example, only modify critical bytes which determine branches. There are mainly two ways to gather information. The first way collects information from past fuzzing runs. For example, AFLFast [6] and FairFuzz [14] are two successful improvers of AFL. AFLFast models the fuzzing process as a Markov chain and assigns modification times to input seeds according to path frequency. FairFuzz collects branch hit count and only modifies inputs which can hit branches whose hit count is small enough. The second way utilizes program analysis techniques. For example, CollAFL [9] and Vuzzer [17] guide fuzzing by control- and data-flow gathered from static analysis. Others integrate symbolic execution to help fuzzing pass the complicated checks [7, 19].

These optimizations are effective and acquire huge performance improvements to the original AFL in single mode[1]. In industry practice, parallel mode of mutation based fuzzers are adopted for higher

*Yu Jiang is the correspondence author.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275525

[1]The single mode means testing the program with instantiating only one fuzzer instance, while the parallel mode means instantiating multiple fuzzer instances at the same time.

efficiency. However, evaluation of those optimizations in parallel mode is hardly observed in empirical statistics or experiment results in literature studies. Hence, we evaluate the performance of AFL, AFLFast and FairFuzz in parallel mode with four fuzzer instances on 12 real-world programs[2].

To our surprise, in parallel mode AFLFast and FairFuzz cover only 97% and 96% branches, and trigger only 57% and 89% unique crashes of AFL. The performance degradation of these optimizations in parallel mode is because these studies lack the consideration for synchronizing their additional guiding information. Thus running multiple instances of those optimized fuzzers in parallel mode yields little performance gain, and sometimes they perform even worse than the original fuzzer. To extend fuzzing optimizations of single mode to industrial parallel mode, we have to face two main challenges:

(1) **Information synchronization mechanism.** Optimized approaches always have additional guiding information, which should be synchronized in parallel mode. A good mechanism needs to spend little resources to synchronize information and make fuzzer instances focus on fuzzing itself.
(2) **Task division mechanism.** Different fuzzer instances in parallel mode tend to perform the similar actions because of the similar guiding information. Dividing the fuzzing task and allocating them to different fuzzer instances promotes the diversity and reduces the waste of resources.

In this paper, we present PAFL, a framework which can extend those existing fuzzing optimizations of single mode to industrial parallel mode. First, we design a synchronization mechanism for additional information synchronization among different fuzzer instances based on a global-and-local data structure. We implement this mechanism with shared-memory and semaphores. Second, we divide the fuzzing task by grouping branches into different parts according to their locations in branch bitmap. Different fuzzer instances are guided to fuzz different parts. We conduct repeated benchmark evaluation using different real-world programs from Google fuzzer-test-suite. Results demonstrate that successful improvers of AFL in single mode such as AFLFast and FairFuzz do not perform as well as original AFL in parallel mode. Augmented with PAFL, the performance of AFLFast and FairFuzz in parallel mode improves, namely covering 8% and 17% more branches, triggering 79% and 52% more unique crashes. For the further evaluation on more real projects from GitHub, optimized fuzzers augmented with PAFL find more real bugs within which 25 have been successfully registered as CVEs in the US National Vulnerability Database.

2 BACKGROUND

Typical Mutation-based Fuzzer: AFL, AFLFast and FairFuzz. Figure 1 illustrates the general workflow of AFL [21]. It maintains an input seed queue built on initial seeds and runs a continuous fuzzing loop. It has four steps: (1) Select input seeds from the queue, (2) mutate the selected input seeds to generate new candidate seeds, (3) run the target program with the candidate seeds, track the coverage and report vulnerabilities (4) add interesting candidate seeds which have new coverage to the seed queue, then go to step 1. Following the continuous fuzzing loop, AFL improves coverage systematically. AFLFast [6] and FairFuzz [14] are two extensions of AFL which collect program running information in step 3 to guide fuzzing to explore more areas. Their improvements focus on step 1 and step 2, namely input seed selection and input seed mutation.

AFLFast collects path frequency and utilizes it in step 1 to prioritize seeds exercising low-frequency paths. It also calculates the number of mutation times based on path frequency in step 2, which assigns more mutation times to seeds exercising low-frequency paths. These improvements avoid wasting energy on hot paths. FairFuzz gathers a more fine-grained information, namely branch hit count. Branch represents a block transition, the fundamental elements of a path. In step 1, it only selects seeds covering branches whose hit count is small. The branch which has the smallest count among covered branches is defined as the target branch. FairFuzz mutates input seeds in a restrict way to ensure the generated seeds still hit the target branch in step 2. AFLFast and FairFuzz perform better than AFL in single mode. However, they lose their advantages in parallel mode, because they lack considerations for synchronizing their necessary guiding information in parallel mode.

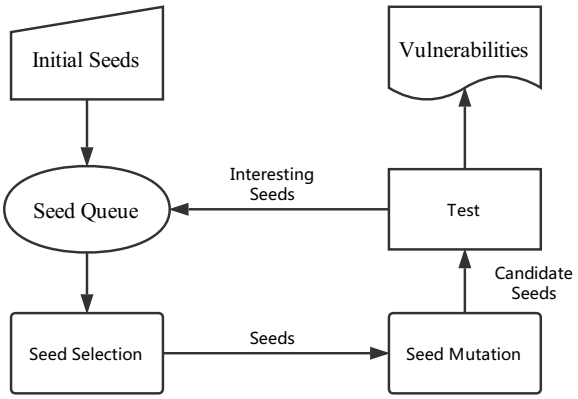

Figure 1: The workflow of AFL.

Parallel Fuzzing. Running fuzzers in parallel mode is common to test the real-world programs. Many widely used industry fuzzers support parallel mode. Most of them, such as AFL [21], libFuzzer [4], honggfuzz [3] coordinate different fuzzing processes by sharing input seeds. Xu et al. propose several new primitives to speed up AFL in parallel mode [20]. They try to shorten the execution time of each iteration of fuzzing. EnFuzz [8] ensembles diverse fuzzers to increase generalization ability. Different from them, we focus on adapting the augmented approaches into parallel fuzzing.In recent years, parallel fuzzing has become a common sense in industry practice. To prevent security incidents, companies and organizations spend lots of resources on fuzzing their products and open source projects. Google's OSS-Fuzz [5], a fuzzing platform, found over one thousand bugs in five months. Besides, Microsoft starts Project Springfield [1], a fuzzing cloud service for developers to test their own works.

3 PAFL DESIGN

To address the challenges and extend single enhanced fuzzing approaches to parallel mode, we propose a parallel fuzzing framework,

[2]Google fuzzer-test-suite: https://github.com/google/fuzzer-test-suite/

namely PAFL . Figure 2 presents the structure of PAFL. Fuzzers run in parallel and synchronize guiding information as well as input seeds regularly. PAFL maintains the global and local guiding information for synchronizing. It also divides the whole fuzzing task into several parts based on branch bitmap. With the guidance, each instance focuses on their own parts and utilizes enhanced fuzzing approaches to improve testing coverage and report bugs efficiently.

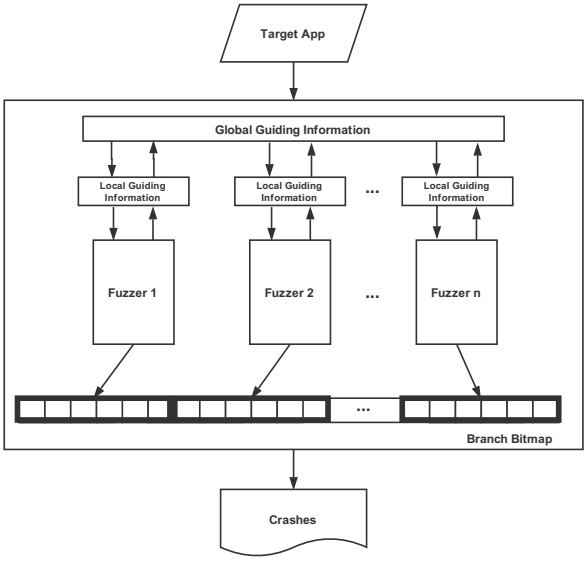

Figure 2: PAFL design.

3.1 Information Synchronization Mechanism

The synchronization mechanism employs a global-local style data structure as shown in Figure 2. The local guiding information is maintained by each fuzzer instance, while the global guiding information is maintained by all fuzzer instances. Each instance has its own local information, so they do not need to synchronize real time. Because fuzzing executes the target program very quickly and the guiding information such as branch hit count may vary rapidly at the early stage, using only one shared global information is also unrealistic. PAFL synchronizes the guiding information and input seeds simultaneously. The synchronization operation consists of three steps:

(1) **Upload step.** Upload local guiding information of each fuzzer instance to global.
(2) **Update step.** Update the global information with the local guiding information.
(3) **Pull step.** Pull the new global guiding information as new local information for each fuzzer instance.

In specific, we take FairFuzz as an example to illustrate the synchronization mechanism in detail. FairFuzz gathers branch hit count to locate rare branches and targets them to increase testing coverage. Because AFL maps branches with a fixed size memory, we also build a fixed size local branch hit count map for each fuzzer instance. Following the synchronizing structure, an equal sized global map is also built to synchronize the hit count.

In the second step, PAFL goes through both the local and global branch hit count map to update the global hit count map. Let $b_{local,i}$ be the hit count of branch i in the local map and $b_{global,i}$ be the corresponding hit count in the global map. The new value for $b_{global,i}$ is:

$$b_{global,i} = \max(b_{local,i}, b_{global,i}) \quad (1)$$

The formula assigns the global branch hit count with the maximum value for all fuzzer instances, which speeds PAFL to identify high frequency blocks in a global scope. The synchronization structure and *max* operation have following advantages: (1) Single fuzzer instances do not need to synchronize at the same time. (2) The global map is as small as local maps and the max operation is cheap, so it is quick and efficient for synchronizing. (3) It is convenient to implement this structure by shared-memory and semaphores. In a similar way, we can customize AFLFast and other fuzzers to PAFL.

3.2 Task Division Mechanism

Different fuzzer instances share similar guiding information and the same guided mutation algorithm, therefore they tend to perform similar actions. This similarity wastes resources and limits the effect of parallel fuzzing. The key to solving this problem is increasing the diversity among different fuzzers. We accomplish this by driving different fuzzer instances focusing on different areas in input seed selection step according to branch hit count. Algorithm 1 shows our solution. For all fuzzer instances numbered from 1 to n, the input seeds assigned to fuzzer m are computed as follows: First, we divide the branch bitmap into n intervals, and the range of bitmap positions corresponding to fuzzer m is computed as line 1-3. Next, given a input seed s, we find its rarest branch which has the smallest branch hit count, as shown in line 4 and 5. If the rarest branch is not in the range, we skip it and try next input seed. Otherwise, we execute normal fuzzing process. By dividing branches and skipping the input seeds which are not in the corresponding parts, different fuzzer instances focus on fuzzing different areas, which promotes the diversity among them and enhances the parallel fuzzing efficiency.

Algorithm 1: Fuzzing Task Division Algorithm

Input : The total number of fuzzer instances: n,
the index of the fuzzer instance: m,
input seed s,
branch hit count map *branch_hit_count*.

1 $interval = MAP\_SIZE/n$;
2 $start = (m-1) * interval$;
3 $end = m * interval$;
4 $branch\_covered$ = `FindCoveredBranch(`s`)`;
5 min_hit_id = `FindMinHitID(`$branch\_covered, branch\_hit\_count$`)`;
6 **if** $min\_hit\_id < start$ *or* $min\_hit\_id >= end$ **then**
7 | Try next input seed;
8 **end**

4 EVALUATION

We implement PAFL on AFLFast and FairFuzz. The synchronizing mechanism is implemented by shared-memory and semaphores. The fuzzing task division mechanism is integrated to the input seed selection step. We evaluate PAFL using 12 different real-world programs of Google fuzzer-test-suite and some other projects from GitHub that have different code scale and application scenarios. Results show that AFLFast and FairFuzz lose their advantage in parallel mode. Augmented with PAFL, their performance recovers as well as in single mode. For practical projects on GitHub, they find much more real bugs, within which 25 are registered as CVEs in the US National Vulnerability Database.

4.1 Google Benchmark Evaluation

We select 12 real-world programs to evaluate PAFL. These typical programs have different code scale and application scenarios, including well-known tools (e.g. libarchive, pcre2, woff2), image processing libraries (e.g. guetzli), communication protocol toolkits (e.g. openssl, boringssl, libssh), color management engines (e.g. lcms), regular expression engines (e.g. proj4, re2), and document processers (e.g. libxml2) etc. Each program is hardened by Google AddressSanitizer (ASan) [18] when compiled.

The number of branches covered and unique crashes triggered are used as metrics. The first metric evaluates the coverage of the target program and the second metric indicates the probability of finding vulnerabilities. It is noteworthy that some crashes may triggered by an identical cause because fuzzers distinguish them by execution paths. However, triggering crashes is the prerequisite for detecting vulnerabilities. The more crashes we found, the higher probability that more vulnerabilities could be identified.

We first evaluate the performance of AFLFast and FairFuzz in parallel mode with four fuzzer instances in 24 hours. Then we augment them with PAFL and repeat the experiments. We collect input seeds generated by all fuzzer instances as initial seeds and reuse AFL in benchmarking-only mode to aggregate results. This mode only feeds the program with initial seeds without mutation to count the metrics. Our experiments are conducted ten times in a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04 as the host OS.

Table 1: Number of branches in parallel mode

Project	AFL	AFLFast	FairFuzz
boringssl	3836	3875	3638
c-ares	105	105	105
guetzli	1522	2324	1514
lcms	2507	2472	2495
libarchive	10582	9267	8646
libssh	604	604	604
libxml2	15269	14204	14351
openssl	4142	4067	4086
pcre2	35219	34725	35124
proj4	324	324	324
re2	16656	16498	16452
woff2	175	175	175
Total	90941	88640	87514

Table 2: Number of crashes in parallel mode

Project	AFL	AFLFast	FairFuzz
boringssl	0	0	0
c-ares	17	12	19
guetzli	0	0	0
lcms	48	88	59
libarchive	0	0	0
libssh	0	0	0
libxml2	63	22	38
openssl	68	36	125
pcre2	3575	2001	3111
proj4	0	0	0
re2	0	0	0
woff2	0	0	0
Total	3771	2159	3352

Table 1 shows the number of branches covered by AFL, AFLFast and FairFuzz in parallel mode. Compared with AFL, we find AFLFast and FairFuzz lose their advantages. In detail, from the second and third column, we observe AFLFast covers 97% branches of AFL in total. Among the 12 programs, it performs better than AFL only in 2 programs. From the second column and the last column, FairFuzz covers 96% branches of AFL in total. It does not perform better than AFL in any programs. Table 2 shows the number of crashes triggered by AFL, AFLFast and FairFuzz in parallel mode. We observe AFLFast triggers 57% crashes of AFL in total, and FairFuzz triggers 89% crashes of AFL in total.

The statistics demonstrate that AFLFast and FairFuzz perform worse than AFL in parallel mode. This is because both AFLFast and FairFuzz depend on additional guiding information to get improvements, however, this information is not shared by fuzzer instances in parallel mode. When they shared the input seeds like AFL, this information is not consistent with other imported seeds. Therefore, instead of augmenting fuzzing, their optimizations become obstacles, which would hamper the performance.

Table 3: Number of branches augmented with PAFL

Project	AFLFast	AFLFast with PAFL	FairFuzz	FairFuzz with PAFL
boringssl	3875	3901	3638	4077
c-ares	105	168	105	168
guetzli	2324	2440	1514	4217
lcms	2472	2677	2495	2706
libarchive	9267	10532	8646	10482
libssh	604	667	604	667
libxml2	14204	17216	14351	22803
openssl	4067	4145	4086	4143
pcre2	34725	37273	35124	36253
proj4	324	327	324	327
re2	16498	16520	16452	16670
woff2	175	177	175	175
Total	88640	96043	87514	102688

Table 3 shows the number of branches covered by AFLFast, FairFuzz and their PAFL augmented versions in parallel mode. From

the second column and third column, we observe that AFLFast augmented with PAFL covers 8% more branches than the original one. From the forth column and last column, we observe that FairFuzz augmented with PAFL finds 17% more branches than original one. When augmented with PAFL, AFLFast and FairFuzz synchronize guiding information among fuzzer instances, which is statistic-based and fits every single fuzzer instance well. Furthermore, PAFL divides the whole fuzzing task into different parts and assigns them to different fuzzer instances. The division increases the diversity among fuzzer instances and make better effects. Synchronizing guiding information and dividing fuzzing task help AFLFast and FairFuzz to cover more branches.

Table 4: Number of unique crashes augmented with PAFL

Project	AFLFast	AFLFast with PAFL	FairFuzz	FairFuzz with PAFL
boringssl	0	0	0	0
c-ares	12	17	19	19
guetzli	0	0	0	0
lcms	88	92	59	97
libarchive	0	79	0	60
libssh	0	0	0	0
libxml2	22	51	38	40
openssl	36	102	125	137
pcre2	2001	3524	3111	4736
proj4	0	0	0	0
re2	0	0	0	0
woff2	0	0	0	0
Total	2159	3865	3352	5089

Table 4 shows the number of unique crashes found by AFLFast, FairFuzz and their PAFL augmented versions in parallel mode. As mentioned earlier, several unique crashes with different triggered paths may have the same cause, but this metric still reflects the probability to detect bugs. For AFLFast in the second column and the third column, after being augmented with PAFL, the bug detection probability increases 79% in total. From the forth column and fifth column, we observe PAFL helps FairFuzz increase 52% bug detection in total. From these comparisons and statistics, we conclude that synchronizing mechanism and fuzzing task division mechanism help AFLFast and FairFuzz hunt more bugs and trigger each bug with a higher probability.

4.2 GitHub CVE Mining

We apply AFLFast and FairFuzz augmented with PAFL to fuzz more realistic projects from GitHub that have been fuzzed before, and they also have a good performance. Besides the coverage improvements, they find more unknown real bugs including 25 successfully registered as CVEs, as shown in Table 5.

We give an analysis of the project libjpeg for a more detailed illustration. libjpeg is a widely used C library for reading and writing JPEG image files. Four previous unknown CVE vulnerabilities in libjpeg contain two segmentation fault, one division by zero error and one logic error. In particular, CVE-2018-11213 allows remote attackers to cause segmentation fault via a crafted file. We analyze the vulnerability with gdb and find it is caused by a memory access

Table 5: The CVE vulnerabilities found by augmented AFLFast and FairFuzz

Project	Count	CVE-Number
cstring	1	CVE-2018-11097
discount	3	CVE-2018-11468, CVE-2018-11503, CVE-2018-11504
lepton	2	CVE-2018-12108, CVE-2018-12109
libjpeg	4	CVE-2018-11212, CVE-2018-11213, CVE-2018-11214, CVE-2018-11813
md4c	4	CVE-2018-11536, CVE-2018-11545, CVE-2018-11546, CVE-2018-11547
PDFGen	1	CVE-2018-11363
ReadStat	2	CVE-2018-11364, CVE-2018-11365
tinyexr	7	CVE-2018-12064, CVE-2018-12092, CVE-2018-12093, CVE-2018-12503, CVE-2018-12504, CVE-2018-12687, CVE-2018-12688
ICMS	1	CVE-2018-12498

Listing 1: CVE-2018-11213

```
Program received signal SIGSEGV,
    Segmentation fault.
0x00000000004f7178 in get_text_gray_row (
    cinfo=0x7fffffffdd80,
    sinfo=0x628000008118) at rdppm.c:153
153     *ptr++ = rescale[read_pbm_integer(
    cinfo, infile)];
```

violation. As the listing 1 shows, the problem is that *ptr attempting to access a restricted area of memory in line 153 of rdppm.c.

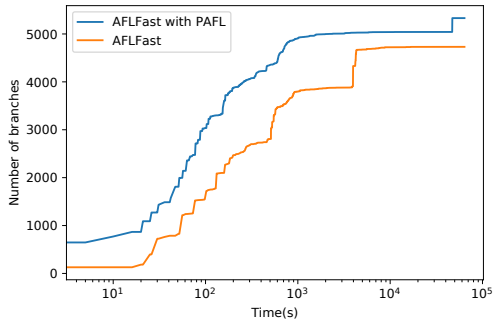

Figure 3: Number of branches over time for AFLFast and its augmented version for fuzzing libjpeg.

We also fuzz libjpeg with original AFLFast and FairFuzz, but they do not find these vulnerabilities. We draw the number of branches covered varying with time for fuzzing libjpeg 24 hours with AFLFast, FairFuzz and their augmented versions in Figure 3 and Figure 4. In Figure 3, PAFL helps AFLFast find more branches with a higher speed than the original one. Figure 4 demonstrates that PAFL augmented FairFuzz covers more branches then the original version. The improved versions all take lead from the beginning to the end. Besides libjpeg, when augmented with PAFL, AFLFast

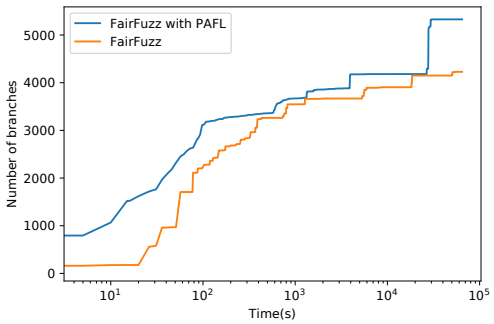

Figure 4: Number of branches over time for FairFuzz and its augmented version for fuzzing libjpeg.

and FairFuzz also perform well on other real-world programs in parallel mode. From these comparisons and analysis, we conclude PAFL extends optimizations of fuzzing approaches in single mode into parallel mode successfully.

5 DISCUSSION

In the developing and practicing of PAFL, we get the following lessons.

(1) **The optimizations in single mode are not easy to take effect in parallel mode directly. It may turn into obstacles in some industrial practice.** The key to improve fuzzing is utilizing additional information to restrict its randomness and guide fuzzing to increase the coverage, as well as the probability to trigger more crashes. If we do not exchange this information between parallel fuzzing instances, only synchronizing input seeds cannot bring the single improvements into global. To reduce the cost of synchronizing, only simple but necessary information is accepted.

(2) **Running parallel fuzzing with input seeds and guiding information synchronizing improves the fuzzing coverage.** In industry environment, companies always have sufficient resources to fuzzing their products. Running parallel fuzzing with input seeds synchronizing is a convenient and scalable way to improve fuzzing.

(3) **The enhanced approaches which extended to parallel fuzzing take effect when different fuzzing instances process different tasks.** When each fuzzer instance gets the synchronized guiding information, they tend to perform similar actions caused by the guided fuzzing algorithm. The similar actions cause several fuzzers running like only one fuzzer. Task division and results merging are two key points a general parallel task focuses on. To take single optimizations take effect in parallel mode, we also need to divide the fuzzing task and assign each to a single fuzzing instance.

6 CONCLUSION

We observe that many optimizations which target to improve fuzzing work well in single mode, but in parallel mode, they fail to maintain the efficiency improvements. In this paper, we propose a framework PAFL which utilizes an information synchronization mechanism and task division mechanism to extend these optimizations to parallel mode. We augment AFLFast and FairFuzz with PAFL and evaluate them on real-world programs from Google fuzzer-test-suite and real projects from GitHub repeatedly. After being augmented, AFLFast and FairFuzz cover more branches and expose more unique crashes in parallel mode than before. They find more real bugs including 25 successfully registered as CVEs. Our future work will focus on developing more efficient synchronization mechanism and adapting parallel fuzzing to distributed programs.

REFERENCES

[1] 2015. Microsoft Security Risk Detection ("Project Springfield"). https://www.microsoft.com/en-us/research/project/project-springfield/. (2015). [Online; accessed 26-January-2018].
[2] 2016. Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. (2016). [Online; accessed 26-January-2018].
[3] 2016. Google. Honggfuzz. http://honggfuzz.com/. (2016).
[4] 2017. libFuzzer in Chrome. https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/README.md. (2017). [Online; accessed 12-November-2017].
[5] 2017. OSS-Fuzz: Five months later, and rewarding projects. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. (2017). [Online; accessed 16-May-2018].
[6] 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, 1032–1043.
[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 380–394.
[8] Yuanliang Chen, Yu Jiang, Jie Liang, Mingzhe Wang, and Xun Jiao. 2018. EnFuzz: From Ensemble Learning to Ensemble Fuzzing. *arXiv preprint arXiv:1807.00182* (2018).
[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. [n. d.]. CollAFL: Path Sensitive Fuzzing. In *CollAFL: Path Sensitive Fuzzing*. IEEE, 0.
[10] Sam Hocevar. 2007. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf. (2007). [Online; accessed 26-January-2018].
[11] Rahul Johari and Pankaj Sharma. 2012. A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. In *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*. IEEE, 453–458.
[12] T Kavitha and D Sridharan. 2010. Security vulnerabilities in wireless sensor networks: A survey. *Journal of information Assurance and Security* 5, 1 (2010), 31–44.
[13] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan. 2011. A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques. In *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*. IEEE, 468–471.
[14] Caroline Lemieux and Koushik Sen. 2017. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. *arXiv preprint arXiv:1709.07101* (2017).
[15] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 562–566.
[16] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).
[17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.
[19] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 61–64.
[20] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2313–2328.
[21] Michal Zalewski. 2015. American fuzzy lop. (2015).