A Quantitative Comparison of Coverage-Based Greybox Fuzzers

Natsuki Tsuzuki
tuzuki@ertl.jp
Nagoya University
Japan

Norihiro Yoshida
yoshida@ertl.jp
Nagoya University
Japan

Koji Toda
Fukuoka Institute of Technology
Japan

Kenji Fujiwara
National Institute of Technology,
Toyota College
Japan

Ryota Yamamoto
Nagoya University
Japan

Hiroaki Takada
Nagoya University
Japan

ABSTRACT

In recent years, many tools have been developed for fuzz testing that generates and executes test cases repeatedly. However, many studies use different fuzzing targets and evaluation criteria and then it is difficult to compare the performance of the existing tools for fuzz testing. Therefore, we prepared a unified collection of fuzzing targets and then compared 8 fuzzers with the benchmark. In comparison, we compared the fuzzers based on the number of execution paths and branch coverage. The result shows that the number of execution paths is significantly different between the fuzzers. On the other hand, the statistical difference is not confirmed between the branch converges of the fuzzers.

CCS CONCEPTS

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

KEYWORDS

Fuzz testing, Coverage, Empirical study

ACM Reference Format:
Natsuki Tsuzuki, Norihiro Yoshida, Koji Toda, Kenji Fujiwara, Ryota Yamamoto, and Hiroaki Takada. 2020. A Quantitative Comparison of Coverage-Based Greybox Fuzzers. In *AST '20: International Conference on Automation of Software Test (AST '20), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387903.3389304

1 INTRODUCTION

In recent years, many tools have been developed for fuzz testing that generates and executes test cases repeatedly. **AFL** [13] is one of the representative fuzzers (i.e., automated tools for fuzz testing) and has a successful track record to discover unknown vulnerabilities. So far, many researchers have developed fuzzers by extending **AFL** [1, 2, 8].

Based on the general scientific manner, a newly-proposed tool has to be compared with the state-of-the-art existing tools. The primary criteria for comparing fuzzers are detection capability (i.e., the number of detected instances, detection time) based on ground truth and code coverage [7]. Although these criteria are commonly-used, the experimental settings based on the criteria may differ between the proposed fuzzers [7]. At first, the measures of coverage may differ such as line or branch coverages. Second, the experimental targets may differ. According to the investigation by Klees et al. [7], *Binutils* is the most commonly-used real-world system for the evaluations of fuzzers. Surprisingly, it was shared by only 4 out of 32 fuzzing-related papers and no overlap when versions are considered.

Generally, when different studies of fuzzers use different experimental targets and/or criteria, it is difficult to compare the experimental results between the fuzzers. Therefore, we prepare a unified collection of fuzzing targets and then compared 8 fuzzers with the benchmark. In comparison, we compared the fuzzers based on the number of execution paths and branch coverage. The result shows that the number of execution paths is significantly different between the fuzzers. On the other hand, the statistical difference is not confirmed between the branch converges of the fuzzers.

The summary of our findings are as follows:

- The newer **AFL**-based fuzzer can execute significantly a larger number of paths (see Section 3.1).
- AFL-based fuzzer improves to branch coverage (see Section 3.2).
- The newer **AFL**-based fuzzer does not always achieve higher coverage (see Section 3.3).

2 RELATED WORK

2.1 Coverage-based fuzzing

Fuzz testing is one of the automated software testing techniques that generate and execute test cases repeatedly. So far, a number of tools (i.e. fuzzers) have developed for fuzz testing.

Fuzz testing can be broadly categorized into blackbox, whitebox, and greybox fuzzing [2]. Blackbox fuzzing analyzes the input structure and the output of programs for generating test cases [10]. On the other hand, whitebox fuzzing [5] analyzes the structure of programs. Greybox fuzzing [2] falls between whitebox and blackbox fuzzing.

AFL is a representative greybox fuzzer [2]. The process of **AFL** is comprised of the following steps: (1) receive initial test cases from an user, (2) generate test cases based on existing test cases, (3) enter the generated test cases into a target software system, (4)

AST '20, October 5–11, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7957-1/20/05…$15.00
https://doi.org/10.1145/3387903.3389304

Table 1: Target fuzzers

Fuzzers	Fuzzing targets	Evaluation criteria
AFLFast	*Binutils 2.26* (*c++filt*, *nm*,*objdump*, *readelf*, *size*, *strings*), *Coreutils 8.25*	Number of Unique Crashes, ground truth (# of detected instances, detection time) line coverage
AFLGo	*Binutils*[a], *Diffutils*, *LibPNG*	Basic Block Coverage, ground truth (# of detected instances, detection time)
FairFuzz	*Binutils 2.28* (*c++filt*, *nm*,*objdump*, *readelf*), *tcpdump*,*xmllint*, *mutool draw*, *djpeg*, *readpng*	Basic Block Transrations Covered, # of occurrence of spefic sequences

[a] no description for specifying the version

gather information (e.g., execution time, executed path) for next test case generation in the execution of the system, and (5) repeat (2), (3), and (4). It identifies paths of executed test cases using the **AFL** coverage profile [7] and then classifies the test cases. After that, it mutates the test cases for generating test cases which falls into a new classification.

In this study, we investigate **AFL** and its extensions (**AFL** (2013), **AFLFast** (2016), **AFLGo** (2017), **FairFuzz**). All of them are categorized as coverage-based fuzzer for detecting crashes by Manes *et al.* [9]. Table 1 shows the targets and criteria for each fuzzer 1.

Böhme et al. showed that **AFL** has the problem of occurring low-frequency paths using a Malcov chain model [2] and then developed **AFLFast** which improves **AFL** so that such paths are executed preferentially. The main differences from **AFL** are prioritization of test cases in a queue and the number of test cases that are newly-generated from selected test cases.

AFLGo is aimed at regression testing after source code modification (e.g., patch adoption) [1]. It is extended from **AFL** to reach and test a user-specified position in the program efficiently. **AFLGo** computes a distance among basic blocks in control flow graphs and then generates more number of test cases when execution traces are near from a user-specified position.

The experiment by Lemieux and Sen shows paths that are not executed by **AFL** [8]. Based on the experiment, they developed **FairFuzz** that is extended **AFL** to execute the paths by mutating the specific part of test cases. In the experiment, Lemieux and Sen applied **AFL** to xmllint in libxml for 24 hours and investigated the line for executing parser.c in xmllint. As a result, **AFL** was unable to test the branch statement to check whether a variable is matched with a specific character sequence. The reason for this limitation is that **AFL** mutates all of the bytes of original test cases in the generation of test cases. To alleviate the limitation, **FairFuzz** performs mutation after masking specific bytes.

2.2 Evaluation of fuzzers

Typical fuzzing research compared a newly-proposed and existing fuzzers to show the usefulness of the newly-proposed one. Unfortunately, the evaluation criteria often differ between **AFL**-based fuzzers. For example, a representative **AFL**-based fuzzer **AFLFast** [2] performed a comparison based on ground truth, detection time, the number of unique crashes, and line coverage. On the other hand, later **AFL**-based fuzzers **AFLGo** [1] and **FairFuzz** [8] are evaluated based on basic block coverage and basic block transitions covered respectively.

According to Klees et al. [7], the evaluations of the existing fuzzers use crash-based and/or code coverage-based criteria. The crash-based criteria are categorized as follows:

- the number of instances in ground truth set
- the number of the groups of related crashes based on stack hashes and/or **AFL**'s notion of coverage profile

Coverage-based criteria include line-level (e.g., line, instruction, basic-block), method-level, and entity-level (e.g., control-flow edge, branch) coverages.

As dataset for the evaluation of fuzzers, several studies have used *Binutils* (as mentioned in Section 1), *LAVA-M* [4], and the test suite of Cyber Grand Challenge[1]. Regarding those datasets, Klees et al. [7] pointed out the following problems:

- Different versions have been used in the case of *Binutils*
- Injected defects into *LAVA-M* [4], and the test suite of Cyber Grand Challenge may not sufficiently resemble defects in the wild.

As shown in Table 1, no common benchmark has been established in the field of fuzz testing. On the other hand, in the field of defect prediction, the common datasets (e.g., Defect4J [6]) including defects in OSS has been frequently-used so far. In the field of search-based software testing, the tool competition was held in SBST 2016 [11]. The tool competition officially announced the evaluation criteria (i.e., statement and branch coverage ratios, fault detection and mutation scores, and preparation, generation and execution times).

As mentioned above, it is difficult to compare the evaluation results of fuzzers because different settings/criteria are used between the fuzzers.

3 EVALUATION

For the comparison of fuzzers, we present the following three RQs.

RQ1 Is a newer **AFL**-based fuzzer able to execute significantly a larger number of paths?
RQ2 Does an **AFL**-based fuzzer improve to branch coverage?
RQ3 Does a newer **AFL**-based fuzzer always achieve higher coverage?

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

Table 2: The number of paths and branch coverage after 6 hours (number of paths / branch coverage [%])

version	program	**AFL 1.94b**	**AFL 2.40b**	**AFL 2.49b**	**AFL 2.51b**	**AFL 2.52b**	**AFLFast**	**AFLGo**	**FairFuzz**
Binutils2.26	*cxxfilt*	5,046/47.2	5,292/47.2	5,817/49.1	5,605/47.2	5,902/49.1	7,487/49.1	5,581/49.1	7,676/49.1
	nm	1,576/19.3	1,817/18.7	1,826/15.4	1,981/18.7	2,233/19.3	1,487/19.3	1,990/19.3	4,291/19.3
	objdump	1,995/22.2	2,240/15.2	2,149/21.3	2,148/14.7	2,158/21.6	2,415/21.9	2,383/21.9	3,750/22.7
	readelf	3,209/13.7	3,199/26.7	3,277/40.0	3,037/41.6	3,123/43.8	3,902/47.7	3,278/46.3	6,783/52.1
Binutils2.28	*cxxfilt*	5,031/47.2	6,194/49.1	5,904/49.1	5,507/49.1	5,649/49.1	7,018/49.1	5,866/49.1	6,553/49.1
	nm	1,579/18.8	1,819/17.3	1,758/18.0	1,736/18.0	1,589/18.8	6,728/18.8	1,679/18.8	3,930/18.8
	objdump	2,099/14.2	2,324/18.7	2,330/16.4	2,059/25.4	2,401/26.9	2,506/27.0	2,093/26.9	4,065/29.2
	readelf	2,100/41.3	2,289/21.2	2,168/25.3	2,151/16.7	2,061/40.7	2,433/45.7	2,158/42.1	6,181/51.1
Binutils2.32	*cxxfilt*	3,546/46.3	3,231/46.3	3,022/44.4	2,954/48.1	3,564/48.1	4,650/48.1	3,366/48.1	5,240/48.1
	nm	1,803/18.9	1,753/19.3	1,838/15.0	1,935/17.9	1,932/19.3	1,993/19.3	1,983/19.3	2,666/19.3
	objdump	2,343/18.9	2,162/25.7	2,370/13.3	2,496/14.9	2,207/21.9	2,736/26.7	2,340/26.2	4,543/29.2
	readelf	3,005/34.8	3,021/12.8	2,929/14.5	2,887/35.8	3,024/38.5	3,794/42.9	2,986/39.7	5,562/44.8

Table 3: Test for significant difference between number of passes and branch coverage after 6 hours (number of passes / branch coverage), an asterisk means p<0.05

	AFL 1.94b	**AFL 2.40b**	**AFL 2.49b**	**AFL 2.51b**	**AFL 2.52b**	**AFLFast**	**AFLGo**	**FairFuzz**
AFL 1.94b	-	/	/	/	/	/	/	*/
AFL 2.40b	/	-	/	/	/	/	/	*/
AFL 2.49b	/	/	-	/	/	/	/	*/
AFL 2.51b	/	/	/	-	/	/	/	*/
AFL 2.52b	/	/	/	/	-	/	/	*/
AFLFast	/	/	/	/	/	-	/	/
AFLGo	/	/	/	/	/	/	-	*/
FairFuzz	*/	*/	*/	*/	*/	/	*/	-

Table 4: Comparison of Branch coverage archived by running the testsuite of Binutils and the fuzzers

target_program	test suite	**AFL 1.94b**	**AFL 2.40b**	**AFL 2.49b**	**AFL 2.51b**	**AFL 2.52b**	**AFLFast**	**AFLGo**	**FairFuzz**
cxxfilt	25.0	51.8	51.8	51.8	51.8	51.8	51.8	51.8	51.8
nm	34.9	41.3	40.5	40.5	40.5	40.5	40.9	40.5	41.7
objdump	48.0	49.4	55.4	49.4	55.6	55.0	55.3	55.2	55.6
readelf	18.9	37.5	38.2	38.4	38.9	36.1	40.1	36.4	42.9

We evaluated **AFL 2.52b** (i.e., the latest version of **AFL**), **AFLFast**, **AFLGo** and **FairFuzz** as well as **AFL 2.51b**, **AFL 2.49b**, **AFL 2.40b** and **AFL 1.94b**. Those 4 versions are the original versions of the existing **AFL**-based fuzzers and/or used in the evaluations of the fuzzers [2][8]. Please note that we do not specify any location of a program to **AFLGo** as same as the other fuzzers. The fuzzing targets are *cxxfilt*, *nm*, *objdump* and *readelf* in *Binutils* which are used in the evaluations of **AFLFast**, **AFLGo** and **FairFuzz**. We used three versions of *Binutils* 2.26, 2.28 and 2.32. The versions 2.26 and 2.28 were used in the evaluation of **AFLFast** and **FairFuzz**. The version 2.32 was the latest version when we performed this evaluation. We used the commands *c++filt*, *nm*, *objdump -d*, *readelf -a* and gave " _Z1fv\n " as the initial seed for *cxxfilt* and the files in the testcases directory of **AFL** for *nm*, *objdump* and *readelf* as same as the evaluation of **FairFuzz** [8]. Each execution of a fuzzer is terminated after exactly 6 hours.

We used the number of paths and branch coverage that are computed by **AFL** as evaluation criteria. Please note that we did not use ground truth for a fair comparison because **AFLFast** has updated after a vulnerability in *Binutils 2.26* was found.

Table 2 shows the branch coverage and the number of paths in this evaluation. Branch coverage is calculated by *afl-cov*.

According to Table 2, we cannot confirm significant difference between **AFL 2.52b**, **FairFuzz**, **AFLFast**, and **AFLGo** in the case of the branch coverage of *cxxfilt* and *nm*. Also, **FairFuzz** outperforms the other fuzzers in terms of branch coverage of *objdump* and *readelf*.

3.1 RQ1: Is a newer AFL-based fuzzer able to execute significantly a larger number of paths?

For a comparison of fuzzers, we performed Steel-Dwass test of the number of paths in Table 2. Table 3 shows the result of the Steel-Dwass test. In the case of the number of paths, we confirmed the statistical difference at 5% significance level between **FairFuzz** and the other fuzzers (except for **AFLFast**). We confirm no significant difference at 5% significance level between any other pair of the fuzzers. **FairFuzz** is the newest fuzzer between the target fuzzers. Compared to **AFL 2.52b** which is the base version of **FairFuzz**, it generated a larger number of paths. For the above results, We answer **RQ1** as follows:

> The newer **AFL**-based fuzzer is able to execute significantly a larger number of paths.

3.2 RQ2: Does an AFL-based fuzzer improve to branch coverage?

Table 4 shows the comparison of branch coverage with and without the fuzzers. In the case with the fuzzers, we used the test suites of the **AFL** as the initial seeds. On the other hand, in the case of without the fuzzers, we simply ran the test suite without fuzzers. Each fuzzer is executed for each test case and terminated after exactly 6 hours.

According to Table 4, running the fuzzers outperforms the running the test suite in the cases of all programs.

For the above results, We answer **RQ2** as follows:

> AFL-based fuzzer improves to branch coverage.

3.3 RQ3: Does a newer AFL-based fuzzer always achieve higher coverage?

We compared the fuzzers between the fuzzers in terms of branch coverage. Using the branch coverages in Table 2, we confirmed no statistical difference at 5 percent significant level between arbitrary pairs of the fuzzers.

For the above results, We answer **RQ3** as follows:

> The newer **AFL**-based fuzzer does not always achieve higher coverage.

3.4 Discussion

The most surprising result of this paper is that the results of the number of paths and branch coverages are different. This implies that different evaluation criteria are possible to lead to different results and findings. Researchers in the field of fuzz testing should choose evaluation criteria carefully to their research purpose.

Also, the results of **RQ1** and **RQ3** imply that newer fuzzers tend to be optimized more for detecting unknown vulnerability. On the other hand, the results imply that newer fuzzers are less optimized for the quality assurance process by increasing branch coverage.

RQ2 implies that the fuzzers are still useful for increasing branch coverage compared to the running of simple test cases in a repository.

We suggest that fuzzers are still useful for quality assurances according to **RQ2**. However, researchers should develop better fuzzers not only for discovering vulnerability but also for the process of quality assurance by increasing branch coverage.

Targets-used in this paper are all related to analysis object files that receive a bunch of bytes as input and perform analysis on byte data. Thus, there is a concern about the bias of the experimental result came from the similar domain.

4 SUMMARY AND FUTURE WORK

In this paper, we performed a quantitative comparison of coverage-based fuzzers. We compared coverage-based fuzzers in terms of the number of paths and branch coverages.

The most surprising result of this paper is that the results of the number of paths and branch coverages are different. According to the result of this study, we suggest that researchers should develop better fuzzers not only for discovering vulnerability but also for the process of quality assurance by increasing branch coverage.

As future work, we mainly evaluated AFL and its variants in this paper, leaving out tools such as Hawkeye [3] which improves directed fuzzing by leveraging a static analysis in its seed scheduling and input generation. We should add other sorts of fuzzers (e.g., Hawkeye) to the comparison. Also, we would like to focus more on fuzzing strategy. Fuzzing tools only provide a black-box phenomenon/observation. We would like to know the insight on why a specific strategy is better in what scenarios [12].

ACKNOWLEDGMENTS

We thank Mr. Ryu Miyaki at Nagoya University for proofreading this paper. This work was supported by JSPS KAKENHI Grant Numbers 17H00731.

REFERENCES

[1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proc. of CCS*. 2329–2344.
[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Trans. Softw. Eng.* (2017), 489–506.
[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proc. of CCS 2018*. 2095–2108.
[4] 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 *Proc. of IEEE S&P*. 110–121.
[5] Godefroid, Patrice and Levin, Michael Y and Molnar, David. 2008. Automated Whitebox Fuzz Testing. In *Proc. of NDSS*, Vol. 8. 151–166.
[6] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of ISSTA 2014*. 437–440.
[7] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proc. of CCS*. 2123–2138.
[8] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proc. of ASE*. 475–485.
[9] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Softw. Eng.* (2019). Early Access.
[10] 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.
[11] Justyna Petke and Gregory Gay. 2016. *The 9th International Workshop on Search-Based Software Testing (SBST 2016).* https://cse.sc.edu/~ggay/sbst2016/ (accessed 2019-12-25).
[12] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin. 2018. Towards Optimal Concolic Testing. In *Proc. of ICSE 2018*. 291–302.
[13] Michal Zalewski. 2013. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. (accessed 2019-08-05).