AFL Extended with Test Case Prioritization Techniques

Gen Zhang and Xu Zhou

***Abstract*—Fuzzing is an efficient testing technique to expose bugs and vulnerabilities, and fuzzers extended with coverage information can generate interesting results and find potential bugs in programs. However, previous coverage-based fuzzers, such as American Fuzzy Lop (AFL), fail to realize the importance of the order of input test cases or they are unable to adopt significant and useful coverage information, so some of them suffer from dramatically poor performance. Meanwhile, the main idea of test case prioritization (TCP) in the field of software testing is to rank the test cases according to a certain rule, helping expose bugs and vulnerabilities. Thus our work concentrates on complementing AFL with the characteristics of TCP and improving the performance of the original AFL.**

In this paper, we present a brand-new fuzzing technique combining essential and practical coverage information and prioritization properties commonly used in TCP, which fundamentally enhancing the process of creating new test cases and finding bugs. We implement our method by extending state-of-the-art fuzzer AFL with TCP techniques and evaluate it on 6 widely-used and open source programs from GNU. We conduct experiments on 6 target programs to illustrate our performance on bug detection. On all of these experiments, improvement of our method is witnessed and significantly better outcomes are generated.

***Index Terms*—AFL fuzzing, test case prioritization, coverage information, software se.**

I. Introduction

Software security is attracting increasing attention these days. Although experts spare no efforts in increasing the reliability of software against security problems, vulnerabilities in software are still common [1]. However, many classes of vulnerabilities, such as functional correctness bugs, are difficult to find without executing a piece of code. With regard to the problem of code executing, there has been much debate about the efficiency of symbolic execution versus more lightweight fuzzers[2]. Symbolic execution tools are able to automatically generate tests with high code coverage on a large set of intricate and complicated programs [3]. While fuzzing is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed inputs.

Symbolic execution is very effective because each test case typically executes the target program along a certain path. However, this effectiveness is the result of a huge amount of constraint solving and program analysis [2]. It triggers a large number of paths in the target program and will result in path explosion. However, with regard to fuzzing, today most vulnerabilities are exposed by particularly lightweight fuzzers that do not leverage any program analysis [4]. For the above reasons, in this paper, we give up classic symbolic execution and focus on extending a stage-of-the-art fuzzer American Fuzzing Lop (AFL).

There are three main types of fuzzing techniques in use: black-box fuzzing [5], white-box fuzzing [6] and grey-box fuzzing [7]. Black-box fuzzing is a technique of software testing without any knowledge of the internal architecture of the target program. It only examines the fundamental aspects of the system, which treats the software as a "Black Box". White-box fuzzing is based on analysis of internal structure of the target program and is very effective and efficient in validating design and assumptions. White-box fuzzing is performed based on the knowledge of how the system is implemented. Grey-box fuzzing tests the program with limited knowledge of the structure of an application. It provides combined benefits of black-box and white-box fuzzing techniques and the tester can design excellent test scenarios.

Since grey-box fuzzing is efficient for real world programs and its combined benifits, it is widely applied in software testing. These methods do give up the time-consuming program analysis, they suffer from low testing accuracy. And we observed that there are two main drawbacks of these grey-box fuzzers: (1) They take test cases as input in the default given order and fail to realize the importance of the order of test cases; (2) They adopt imprecise and low-accuracy coverage information for the fuzzing loop, thus existing grey-box fuzzers have been effective mainly in discovering superficial bugs, close to the surface of software, while struggling with more complex ones. On top of all these reasons, we focus on extending state-of-the-art grey-box fuzzer AFL.

AFL is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorith-m. And in previous works, AFL is proved to have high accuracy and able to reveal deeper bugs in programs [2]. However, the process loop of AFL uses very rough and imprecise coverage information to guide the fuzzing process:

1) AFL measures a certain block transition (or a branch) with a ˆ (XOR) operation of current execution location CurLoc and previous execution location PrevLoc. However, it simply adopts randomly generated values to represent current location CurLoc, which is imprecise and may cause duplicate locations.
2) And AFL determines a new state by checking the change of branch bitmap after a certain execution using an & (AND) operation. However, this kind of method cannot measure the extend of the change in branch bitmap coverage. And the above 2 drawbacks of AFL makes it very hard to provide precise coverage information for the fuzzing loop. And as discussed above, similar to

Manuscript received October 30, 2017; revised December 12, 2017.

The authors are with the College of Computer, National University of Defense Technology Changsha, Hunan, China (e-mail: zhanggen12@hotmail.com, zhouxu@nudt.edu.cn).

DOI: 10.7763/IJMO.2018.V8.622

other grey-box fuzzers, (3) AFL fails to realize the importance of the order of the test cases. Enlightened by the idea of TCP, in this paper, we integrate useful prioritization properties and coverage measurement widely used in TCP into AFL to enhance the process of generating coverage information, selecting test cases and finding bugs. On all of our experiments, performance of our method is witnessed and significantly better outcomes are generated.

The main contribution of this paper is as follows:

• We aggregate specific and fundamental prioritization properties of TCP into AFL and we adopt more practical and precise coverage information widely used in TCP to guide the generation of more interesting test cases.

• We implement our proposed method by extending a state-of-the-art grey-box fuzzing tool, American Fuzzy Lop, which is efficient in processing and easy for implementation.

• We evaluate our proposed model on 6 real-world applications and conduct intensive comparison experiments. Experiment result shows that our method outperforms traditional fuzzers with regard to the ability to find bugs and detecting vulnerabilities.

II. Background

A. American Fuzzy Lop

Coupled with a significantly simple but sound coverage-guided fuzzing algorithm, American Fuzzy Lop (AFL) is a well-known fuzzing tool. It adopts a instrumented edge coverage to detect new changes in the program execution trace [8].

The overall algorithm can be summed up as:

• (1) Load starting test cases into the fuzzing queue,

• (2) Take next test case from the fuzzing queue,

• (3) Try to minimize the test case to the smallest size that doesn't affect the program trace,

• (4) Mutate the test case using a well-designed mutation strategies,

• (5) If any of the mutations resulted in a new program trace, add this mutation as a new entry in the fuzzing queue.

• (6) Go to (2).

AFL does not focus on any acknowledged principle and it is not designed according to any theory. While AFL can be thought of as a collection of effective and practical hacks. And it has been implemented in the simplest, most rock-solid way [4]. Thus extension on AFL can give us a high level platform to start with.

B. White-, Grey- and Black-box Fuzzing Techniques

Fuzzing techniques can be classified according to the knowledge acquired from program. Typically, white-box fuzzer has full information of the target program and can use traditional program analysis techniques to uncover properties of the target. White-box fuzzers including SmartFuzz [9], BuzzFuzz [10] and Vuzzer [11] achieve expected performance and can be applied to real-world scenarios. While grey-box fuzzer uses specific feedback information to enhancing the process of "blind" fuzzing. This kind of fuzzing tries to maintain the simplicity of black-box while improving the effectiveness of fuzzers by adopting additional information. AFL and AFLFast [2] are the most successful representation of grey-box fuzzers. Meanwhile, black-box fuuzer does not have any information of the target program at all. Recently new ideas are put into black-box fuzzers and Radamsa [12], zzuf [13] and peach [14] did remarkable work in this field.

This kind of classification of fuzzers are based on the interaction with the target program, while it can also be classified into non-kernel and kernel fuzzers depending on weather it can be used to fuzz kernels [15].

C. Test Case Prioritization Techniques

Elbaum et al. [16] first defined the test prioritization problem. Assuming that $P(T)$ denotes the set of permutations of a given test suite T, and f denotes a function from $P(T)$ to real numbers, a TCP problem is to find $T \in P(T)$, that $\forall T'', T'' \in P(T) \bigwedge T'' \neq T' \Rightarrow f(T') \geq f(T'')$ [17]. In the field of TCP techniques, total and additional schedules are recognized as representative test prioritization techniques due to their effectiveness and are taken as the control techniques in the evaluation of existing work [18]. Meanwhile, most existing TCP techniques guide their prioritization process based on coverage information, which refers to whether any structural unit is covered by a test. TCP techniques can be distinguished by the source code elements they seek to cover: statements (S), branches (B) and methods (M) [19]. Moreover, previous work has already shown that statement-level coverage is at least as effective as other coverage types [20].

Test case prioritization is an important field of software testing and debugging. First introduced by Rothermel et al. [21], it mainly focuses on various strategies or algorithms used in test prioritization. The most widely used TCP techniques are total and additional strategies [21]. By extending total and additional strategies, various of related work are presented these years. Li et al. [18] first uses the idea of search algorithm to solve the TCP problems. And Jiang et al. [22] proposed the famous adaptive random test (ART) prioritization, which selects a test case farthest from the already selected tests. While Nguyen et al. [23] presented an algorithm to prioritize tests based on information retrieval (IR). Furthermore, Saha et al. [24] also proposed a more advanced technique, named REPiR, based on IR.

TCP techniques can be classified into different criteria based on different standard. Henard et al. [19] conducted an intensive experimental study on white-box TCPs and black-box TCPS. While Luo et al. [17] compared the difference between dynamic TCPs and static TCPs.

III. Experiment Setup

A. Adopted Methodology

In this section, we will discuss the TCP techniques adopted in our proposed method. According to previous works [19] in this field, Additional Branch (AB), Additional Spanning Statements (ASS) and Additional Spanning Branches (AS-B) produce the best detection results in source-code-aware techniques. While Input Test Set Diameter (I-TSD), *t*-wise

(*t*-W) and Input Model Diversity (IMD) achieve the highest detection rate in source-code-agnostic techniques. The details of the above techniques are as followed:

- Additional Branch: Covering the maximum number of uncovered branches [25].
- Additional Spanning Statements: Covering the maximum uncovered dominating statements [26].
- Additional Spanning Branches: Covering the maximum uncovered dominating branches [26].
- Input Test Set Diameter: Maximizing the NCD distance between multi-sets of inputs [27].
- *t*-wise: Covering the maximum interactions between *t* model inputs [28].
- Input Model Diversity: Maximizing the Jaccard distance between model inputs [29].

B. Tools and Target Programs

We compile the target programs with gcc 4.7.1 and acquire statement-, branch- and method- coverage information of all initial test cases with the built-in gcov tool [30].

As for the target programs, following previous works, we select 6 real-world and open source programs, Bash, Flex, Grep, Gzip, Make and Sed, which are available online.

IV. Model Overview and Implementation

In this section, the whole process of our method will be presented and we will describe our proposed model in detail. As shown below, generally our proposed model consists of these separate parts:

- (1) At first step, we rank the initial test cases with the techniques mentioned in Section III-A to form a queue of inputs.
- (2) Later, function flip bit() is processed to generate different mutations originate from one previous test case.
- (3) And we still follow the design of AFL and function common fuzz stuff() is processed to perform a single loop of fuzzing with the inputs in the queue.
- (4) We abandon the imprecise coverage measurement in AFL and modify has new bits() with our coverage measurement. Mutations covering new areas of programs will be added to a new entry in the queue to Step (2) and processed for a new loop of fuzzing.
- (5) Go to (2).

In the following sections, we will discuss the additional and modified functions of our model in detail.

A. Rank the Initial Test Cases with Prioritization Techniques

This prioritization step will form a queue Q_I of ordered test cases as inputs for the first run. In this step, we will adopt the three source-code-aware techniques (AB, ASS and ASB) and three source-code-agnostic techniques (I-TSD, *t*-W and IMD) separately, generating $Q_{P_{AB}}$, $Q_{P_{ASS}}$, $Q_{P_{ASB}}$, $Q_{P_{I\text{-}TSD}}$, $Q_{P_{t\text{-}W}}$ and $Q_{P_{IMD}}$ accordingly. This kind of experiment design will lead to 6 sub-experiments and we will discuss the results of them in detail in Section V.

B. Start Fuzz and Detect New Behavior

This part of our model mainly complete the fuzzing process in common fuzz stuff() and detect new behavior of the target program in has new bits(). The basic idea of this part is to detect weather a test case achieve new areas of the target program. And the original AFL uses rough and imprecise method to measure the coverage information. This will lead to false positives in the fuzzing loop and may retain the useless test cases while giving up the interesting ones. However, our proposed model measure coverage information of the target with precise and compile-level tool *gcov*. The recordings of gcov actually reflect the behavior of the running program precisely and this kind of implementation will not give away any useful test cases in the queue. And thanks to the mechanism of gcov (it only requires compiling the target program with gcov parameters for once), we are able to insert gcov into AFL while maintaining the properties of AFL.

We implement our code coverage measurement with gcov, and the difference between the original version and our modification is that we abandon the coverage results of the original method as described in Section I and we use coverage results of gcov in our modification. Our modified function has new bits() basically calculates the coverage information of a test case and decides whether it brings new behavior by comparing it to history coverage information. Test cases with new behavior will be retained and pushed back to the queue for later mutation and useless ones will just be abandoned.

In addition, basic procedure of our model is illustrated in Algorithm 1.

Algorithm 1 Our Proposed Method of Enhancing Fuzzing

Input:
Q_I: Queue of initial test cases
C: Input coverage information acquired through gcov
M : Input model information
P : Target program
Output:
BV : Potential bugs and vulnerabilities
Procedures:

1: Load initial test cases Q_I, coverage information C and model information M, and compile target program with gcov parameters
2: Rank the initial test cases Q_I with the techniques mentioned in Section III-B to form a queue of inputs Q_{P*} (one of $Q_{P_{AB}}$, $Q_{P_{ASS}}$, $Q_{P_{ASB}}$, $Q_{P_{I\_TSD}}$, $Q_{P_{t\_W}}$ and $Q_{P_{IMD}}$)
3: Function flip bit() is processed to generate 16 different mutations [31] Q_M originate from one previous test case
4: Process common fuzz stuff() with Q_M to perform a single loop of fuzzing
5: Run has new bits() to decide weather a mutation covered new areas of the program and add new ones to the queue
6: Go to 3 until bugs and vulnerabilities BV are exposed

V. Experimental Results

In this section, we will present the experiment results of our method and discuss the improvement over the original version of AFL. We ran our experiments on an Ubuntu 14.04 LTS system equipped with a 64-bit 8-core Intel CPU and 32 GB RAM. And 6 target programs from GNU are included: Bash, Flex, Grep, Gzip, Make and Sed. The experiments are aimed to illustrate the ability to detect potential bugs and

vulnerabilities of our model.

For each of our target programs, we measure the performance of the 6 TCP techniques with the original AFL. And we run each experiments on our host for 24 hours to get the results. We illustrate the results of unique crashes in Table I. In the definition of AFL, unique bugs is the main indicator of the ability to detect potential bugs and vulnerabilities. As shown in Table I, the first column contains the 6 target programs described in Section III-B and the second to seventh column are the 6 TCP techniques described in Section III-A. While the last column is the performance of the original version of AFL. In the same time, each single experiment is conducted 20 times in order to get the values that occur the most frequently (mode) to avoid the effect of other factors. However, due to lack of input model data, experiments on the row Bash and column *t*-W, IMD cannot be conducted. And they are marked as "-".

Furthermore, as shown in Figure 1, we collect crashes over time in Gzip for our 6 proposed methods compared with AFL. We can see that after 24 hours execution, the curve of AFL is already flat, while our proposed 6 methods all have an upward trend, which means our proposed model is able to reveal deeper bugs and branches hidden in the program that AFL cannot.

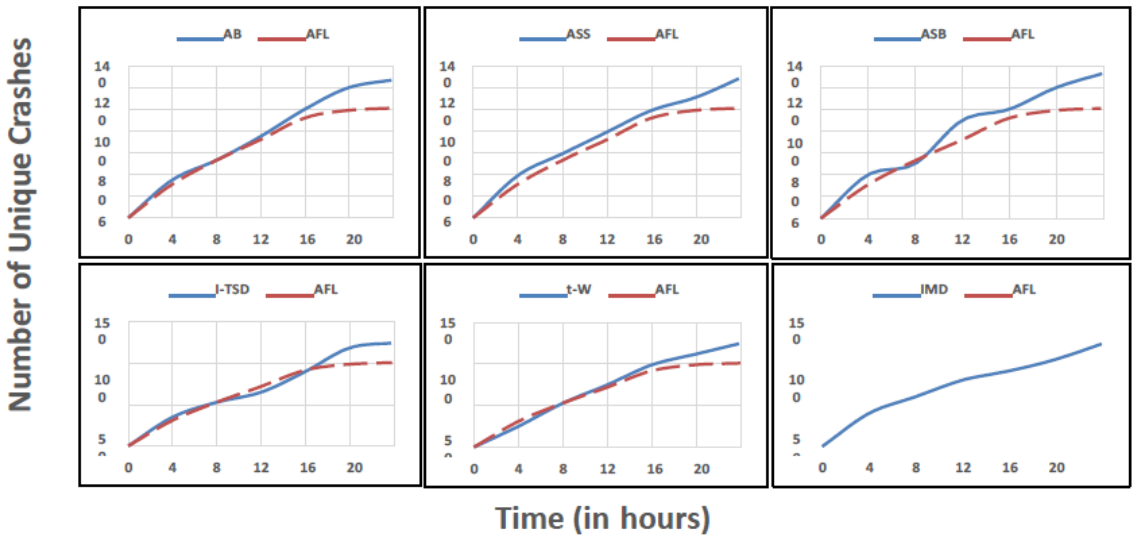

Fig. 1. Crashes over time in Gzip for our 6 proposed methods (solid line) vs. AFL (dashedline).

As shown in Table I, three obvious conclusions can be made. First, AFL extended with the 6 TCP techniques outperforms the original AFL in all experiments, even in cases like Make, where the original AFL cannot find any unique crashes. In terms of unique crashes exposed, our proposed model presents more potential ability in detecting bugs and vulnerabilities. Then, according to the table, ASB catches the most unique bugs in 4 out 6 target programs. But compared to other techniques, the difference in unique crashes is small. At last, in all test cases, source-code-aware techniques outperforms source-code-agnostic techniques in detecting bugs and vulnerabilities. This is in accordance with common sense that source-code-aware techniques is able to adopt more information in the target programs. Furthermore, Table II shows the maximal and minimal result of our 20 times experiments.

This is the Mode Result of Our 20 Times Repea ted Experiments and Mode Means the Value Occurs the Most Frequently in Them.

TABLE I: PERFORMANCE ON DETECTING BUGS OF OUR MODEL

Name	Mode of unique crashes						
	AB	ASS	ASB	I-TSD	*t*-W	IMD	AFL
Bash	291	301	**320**	290	-	-	255
Flex	71	**75**	71	70	69	68	58
Grep	80	79	**81**	78	79	75	61
Gzip	127	129	**133**	125	125	124	101
Make	40	**43**	41	40	39	39	0
Sed	93	95	**101**	91	90	91	79

TABLE II: PERFORMANCE ON DETECTING BUGS OF OUR MODEL

	Minimal and maximal of unique crashes						
	AB	ASS	ASB	I-TSD	*t*-W	IMD	AFL
Bash	285/296	294/307	**311/322**	286/293	-	-	249/258
Flex	67/73	**69/77**	68/73	68/71	65/69	64/69	55/58
Grep	77/82	74/80	**79/83**	75/79	75/79	72/77	59/62
Gzip	121/129	123/130	**131/135**	119/126	120/125	119/125	95/105
Make	36/42	**38/45**	37/42	37/42	35/40	35/39	0/0
Sed	85/95	88/97	**95/104**	85/93	85/90	86/92	71/81

This is the Minimal and Maximal Result of Our 20 Times Repeated Experiments.

In conclusion, our proposed method outperforms the original AFL both in the ability to detect bugs and time cost. The internal reason is that, as described above, AFL fails to adopt a precise coverage measurement method and it uses rough and unrepresentative coverage information from the target program. While our model aggregate TCP techniques into AFL and avoids the above drawbacks by modifying these unwell-designed parts of AFL and achieve a relatively better performance.

VI. Conclusion

In this paper, in all the implementation of fuzzing, we concentrate state-of-the-art AFL to make extension for its promised characteristics. Furthermore, in order to get rid of the internal drawbacks of AFL, we combine the properties of AFL and test case prioritization techniques and aim to achieve better performance. And we present a brand-new fuzzing technique combining essential properties and practical coverage information commonly used in TCP, which fundamentally enhancing the process of creating new test cases and finding bugs. And by conducting experiments on 6 target programs to illustrate our performance both on detecting bugs, improvement of our method is witnessed and significantly better outcomes are generated.

Acknowledgment

The work is supported by The National Key Research and Development Program of China (No. 2016YFB0200401).

References

[1] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. the Network and Distributed System Security Symposium*, 2016.

[2] M. Bo ̈hme, V. T. Pham, and A. Roychoudhury, "Coverage- based greybox fuzzing as markov chain," in *Proc. the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1032–1043, ACM, 2016.

[3] C. Cadar, D. Dunbar, D. R Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI*, vol. 8, pp. 209–224, 2008.

[4] M. Zalewski. American fuzzy lop (afl) fuzzer-technical details. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical details.txt. Accessed: September 1

[5] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Pearson Education, 2007.

[6] P. Godefroid, M. Y Levin, D. A Molnar *et al.*, " Automated whitebox fuzz testing," *NDSS*, vol. 8, pp. 151–166, 2008.

[7] M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 3, no. 6, 2012.

[8] M. Zalewski. American fuzzy lop (afl) fuzzer-readme. [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt

[9] D. Molnar, C. L. Xue, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proc. Conference on Usenix Security Symposium*, pp. 67–82, 2009.

[10] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE International Conference on Software Engineering*, pp. 474–484, 2009.

[11] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-Aware Evolutionary Fuzzing*, 2017.

[12] A. Helin. A general-purpose fuzzer. https://github.com/aoh/radamsa. Accessed: September 1, 2017.

[13] Sam Hocevar. zzuf. [Online]. Available: https://github.com/samhocevar/zzuf

[14] P. Tech. Peach. [Online]. Available: https://www.peach.tech

[15] Thgarnie. Syzkaller. [Online]. Available: https://github.com/google/syzkaller

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," *Software Engineering IEEE Transactions*, vol. 27, no. 10, pp. 929–948, 2000.

[17] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proc. ACM Sigsoft International Symposium on Foundations of Software Engineering*, pp. 559–570, 2016.

[18] Z. Li, M. Harman, and R. M. Hierons, " Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[19] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *Proc. International Conference on Software Engineering*, pp. 523–534, 2016.

[20] H. Mei, D. Hao, L. M. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.

[21] R. Gregg, U. Roland, C. Chengyun, and H. M. Jean, "Test case prioritization: An empirical study," pp. 179– 188, 1999.

[22] B. Jiang, Z. Y. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–244, 2009.

[23] C. D. Nguyen, "Alessandro marchetto, and paolo tonella, test case prioritization for audit testing of evolving web services using information retrieval techniques," in *Proc. IEEE International Conference on Web Services*, pp. 636–643, 2011.

[24] R. K. Saha, L. M. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proc. International Conference on Software Engineering*, pp. 268–279, 2015.

[25] S. Elbaum, A. G Malishevsky, and G. Rothermel, " Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[26] M. Marr and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 974–984, 2003.

[27] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases,"pp. :223–233, 2015.

[28] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[29] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test con- figurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[30] GNU. Gcov. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[31] M. Zalewski. Afl fuzzing strategies. [Online]. Available: http://lcamtuf.blogspot.com/ 2014/08/binary-fuzzing-strategies-what-works.html

Gen Zhang was born in China, 1993. He is a current master student in College of Computer, National University of Defense Technology, Changsha, Hunan, China, 410073.

His major is in computer science and software analysis, and he got his bachelor's degree in 2016 and is going to get his master's degree in 2018.

He published one paper, *Similarity based Matrix Factorization for Recommender Systems*, in ISCID 2017. His current research interests are fuzzing, software analysis and binary analysis.

Mr. Zhang received Extraordinary Student Awards in 2015 in College of Computer for his good performance.

Xu Zhou was born in China, 1985. He is an assistant researcher in College of Computer, National University of Defense Technology, Changsha, Hunan, China, 410073. His major is in computer science and parallel, and he got his doctor's degree in 2013 in NUDT.

He published papers on *PPoPP* and several top transactions on parallel and his research interests are computer system and parallel. Mr. Zhou is a fellow member of IEEE.