Research on Directed Grey-box Fuzzing Technology Based on Target

Liang Sun
National University of Defense Technology
Changsha, China
E-mail:nirvana_sl@126.com

Peng Wu
National University of Defense Technology
Changsha, China

Shaoxian Shu
Hunan Institute of Traffic Engineering
Hengyang, China

***Abstract*—In recent years, grey-box fuzzing has been proven to be the most effective method for discovering vulnerabilities in software. However, the present grey-box fuzzing still has some shortcomings. Most existing grey-box fuzzers are coverage guided and consider the program code equally, and spend a lot of time on improving the code coverage. However, most of the code in the program does not contain bugs and only a small percentage of the code may have bugs. Therefore, blindly improving code coverage can waste limited resources on a large number of bug independent locations and reduce the efficiency of fuzzing.**

In order to solve the above problems, we propose a targeted mutation strategy for continuous target exploration. By identifying the key bytes in the input seeds, a mutation algorithm for different stages of mutation is proposed to ensure that the subsequent generation of seeds can still hit the target location as much as possible, to realize the continuous exploration of the target location. In addition, based on this mutation strategy, we propose a fuzzing optimization method based on multi-factor seed selection, using LLVM framework to insert the target location information into the test program for preprocessing, and then the multi-factor seed selection strategy is used to select better seeds to explore the target location.

Keywords-fuzzy testing; directed grey-box fuzzing; seed mutation strategy; seed selection strategy

I. INTRODUCTION

Vulnerability mining techniques have received a lot of attention due to the increase in the number of vulnerabilities and the intensification of the damage. Among the many vulnerability mining techniques, fuzzy testing techniques [1] have been proven to be one of the most effective techniques for detecting software security[2, 3], which was first proposed by Miller et al. in 1990. It can be generally divided into white-box fuzzy testing, black-box fuzzy testing, and gray-box fuzzy testing [4] which gray-box fuzzy testing has been proven to be efficient and effective.

There are also some issues in gray-box fuzzy testing. American Fuzzy Loop (AFL) [5], the most representative gray-box fuzzy testing tool in the industry, which does not differentiate the code in the process of fuzzy testing and does not guide the direction of variation, which makes the variation of AFL random and blind.

Based on the widely used AFL tool, this project identifies the high-risk target locations in the program to be tested through manual analysis or static analysis reports, selects the seeds that can easily hit the target locations during the fuzzy testing phase, and "controls" the seed variants so that the subsequent variants can still reach the target locations as much as possible.

DOI reference number: 10.18293/SEKE2023-182

II. RELATED WORKS

Directed fuzzy testing is a vulnerability detection technique for target locations in a user-specified program. Unlike coverage-oriented fuzzy testing, directed fuzzy testing spends a lot of time exploring a given target location in the code rather than wasting a lot of time on irrelevant areas of program code. Most existing directed fuzzy testing tools are based on symbolic execution, which uses directed symbolic execution techniques to transform the reachability problem of reaching a target location into an iterative constraint solving the problem for the ultimate purpose of directed fuzzy testing. The effectiveness of directed symbolic execution comes at the cost of efficiency, which spends a significant amount of time on program analysis and constraint solving. In each iteration, directed symbolic execution uses program analysis to determine which paths better approach the target location, constructs corresponding path conditions based on the sequence of instructions along those paths, and uses a constraint solver to check the satisfiability of those conditions.

Directed gray-box fuzzy testing (DGF) is a vulnerability detection technique based on gray-box fuzzy testing to implement directed DGF, which retains the efficiency of gray-box fuzzy testing and usually leaves all program analysis in the program compilation phase. Once the target location is marked, DGF needs to generate seed inputs to reach the target location. In addition to marking the target locations, researchers have also noticed that the interrelationships between targets also help to reach the targets.

V-Fuzz[6] is oriented to vulnerability probabilities, and deep learning models predict vulnerability probabilities to guide the fuzzing process to potentially vulnerable code regions. semFuzz[7] can automatically recover knowledge related to vulnerabilities from text reports and use this information to guide the system in building test cases to trigger known or related unknown vulnerabilities. Seed inputs that perform well in DGF can bring the fuzzy testing process closer to the target location and improve the performance of the subsequent mutation process. Studies have shown that the classical directed fuzzy testing tool AFLGo[8] generates a large number of inputs that are unable to reach the location of the code with vulnerabilities . Therefore, optimizing input generation can be of great help to improve the directionality of DGF. seededfuzz improves the initial seed generation and selection for directed fuzzing. It uses static analysis, dynamic monitoring, and symbolic execution techniques on the target program to select and generate appropriate seeds for directed fuzzing, it identifies the vulnerability-sensitive parts of the input seeds, generates new inputs by changing the relevant bytes, and feeds them to the target program to trigger exceptions.FuzzGuard[9] uses a deep learning-based approach that filters out inputs that cannot reach the target location before learning, and uses a large number of inputs marked as reachable to train the model. The model is then used to predict the likelihood that the newly generated inputs will reach the target location without running them directly, thus saving time on the actual execution.SemFuzz uses information (system calls and parameters) retrieved from CVE descriptions and git logs to generate seed inputs to increase the probability of hitting the vulnerability function.TIFF and ProFuzzer identify input types to help mutate and maximize the likelihood of triggering memory corruption errors.

Selecting seeds that better hit the target location is a key part of DGF. AFLGo generates the call graph and control flow graph of the program to be tested at compile time to calculate the distance of the seed to the target basic block, prioritizing seeds that are close to the target location. RDFuzz[10] combines distance and frequency to prioritize the seeds. One drawback of the distance-based approach is that it focuses only on the shortest distance, and when there are multiple paths to the same target, seeds that are farther away may be ignored.

III. Target-guided Directed Gray Box Fuzzy Testing Methods

Most existing gray-box fuzzy tests are code coverage guided [2], where the most classic coverage oriented tool, AFL, is designed to find more vulnerabilities by increasing the code coverage. In general the broader the scope of code covered during the execution of fuzzy tests, the more likely it is to find potential vulnerabilities in the code. However, most areas of code in real software are not vulnerable, and only some areas may be vulnerable. Therefore, all code should not be treated equally, and more resources should be given to code locations with higher risk or user requirements to explore, instead of wasting a lot of time and computational resources in unrelated code areas, which leads to a decrease in the efficiency of fuzzy testing. Yet there is randomness and blindness in the seed variation part of fuzzy testing that generates a large number of inputs. This leads to the fact that it is often difficult for fuzzy tests to ensure that the mutated inputs can still reach the target location. In particular, when encountering some branches composed of complex conditions, it is more likely to be difficult to reach the target position, and even if the target position is occasionally hit, it is difficult to reach it again. Moreover, most of the vulnerabilities that exist in actual software are not of the type that will be triggered simply after execution, which requires fuzzy testing tools not only to generate seed inputs that can reach the target location, but also to generate a large number of inputs that can continuously reach the target location after seed mutation to achieve continuous exploration of the target location.

A. A Goal-Oriented Directed Gray-Box Fuzzy Testing Framework

This paper improves on the classical fuzzy testing tool AFL by improving the source code preprocessing, seed selection strategy and seed variation method, and proposes the goal-oriented directed gray-box fuzzy testing tool DTFuzz. It can be seen that the entire directed gray box fuzzy testing framework is divided into two phases: the preprocessing phase and the fuzzy testing cycle phase, in which the fuzzy testing cycle mainly includes the multi-factor seed selection strategy and the directed mutation strategy.

After obtaining the source code of the program to be tested, the first thing that needs to be determined is which locations are the target locations in this test. This part can be analyzed by program static analysis tools such as Cppcheck, Clang, etc., and the generated results are processed to generate a uniform format target file, or the target location information can be determined by the user directly specifying the target location.

Next, we move to the fuzzy test loop section, where the inputs are the staked binary program, the initialization seeds, the target location information (actually present in the staked program), and the outputs of the fuzzy test loop phase are the seeds that cause abnormal program behavior (e.g., crashes or timeouts). The priority queue of seeds is obtained through a multi-factor seed selection strategy giving higher variant priority to the preferred seeds. This multi-factor seed selection strategy first distinguishes between seeds that hit the target location and seeds that do not. If the seed does not hit the target, it will use the original AFL strategy, and if the seed can hit the target location, it will have a higher priority. Finally, there is the targeted mutation strategy for the continuous exploration of the target, which can be partly generated not only by mutation DTFuzz first mutates each byte of the seed (using AFL's deterministic mutation phase method) to run the program and determine whether the mutated seeds can hit the target location. For those seeds that can hit the target location, DTFuzz records the location and the mutation method used to mutate these seeds during the mutation process.

B. Targeted Variation Strategies for Targeted Continuous Exploration

The various strategies of both the gray-box fuzzy testing tool AFL and the directed fuzzy testing tool AFLGo do not take into account the use of existing seed variation information, and each variation is performed randomly. AFL's fuzzy testing strategy is to increase the coverage of the code as much as possible, and when the seeds hit the target location, AFL marks them as executed, giving them less energy for subsequent seed prioritization and energy scheduling, and it will explore other branches to discover new paths.

To solve this problem, this paper proposes a target continuous exploration(TCS) for target continuous exploration. Firstly, the key nodes that affect the input seeds to reach the target location are identified by analysis, and it is recorded which variation can still hit the target location, so that in the subsequent variation process, the mutated seeds can be guaranteed to hit the target location area as much as possible, thus achieving the continuous exploration of the target location and improving the efficiency of the directed fuzzy test.

C. *Keyword Section Determination*

First of all, it should be clear that the directional variation proposed in this chapter is used only for seeds that can hit the target location. The variation strategy is still used for seeds that fail to hit the target location, while the original AFL variation strategy is still used for seeds that fail to hit the target location.

What prevents fuzzy tests from reaching the target location is usually caused by conditional statements that may be directly related to some bytes of the input or obtained through a series of complex operations. Analyzing the relationship between the input and the conditionals can cause a lot of overhead and thus reduce the performance of the fuzzy test. Efficiency, if the mutated seed can still hit the target location during the execution, by recording the location where this mutation occurs and the corresponding mutation mode, the connection between the input and the conditional branch can be established to some extent, to achieve the purpose of continuous exploration of the target.

D. *Directional Variation*

After determining the input keyword stanza it goes to the mutation phase of the fuzzy test. For deterministic mutation, not only one byte at a time is mutated, but the step size of each mutation is continuously increased. Obviously, for single-byte mutations, the mutation can be performed directly based on the content of the record, while the situation is slightly different when the mutation step is multiple bytes. Multi-byte mutations can only be mutated if all the bytes they cover can be mutated in the same way, otherwise they are not mutated.

When random mutation is performed, the mutation method used for this mutation is determined first (mutation methods using a combination of mutation operations are considered in order), and the range of bytes that can be mutated can be calculated based on the previously recorded information, so that when the mutation range is randomly selected, the subrange of the mutable byte range can be selected. If such a variation range cannot be found this variation is skipped to proceed to the next cycle of the random variation stage.

By processing the deterministic and random mutation stages as described above, it is possible to generate as many seeds as possible in the mutation stage to reach the target location, especially for those seeds that have already hit the target location, and to mutate the other bytes of the seeds without affecting their hitting the target location again, so that the target location can be fully explored continuously.

TABLE I. CONTINUOUS EXPLORATION ABILITY OF TARGETS AFTER VARIATION STRATEGY IMPROVEMENT

Actual Software	**Target hits**	
	AFLGo	**TCS**
readelf	44.7/sec	46.4/sec
xmllint	65.9/sec	77.1/sec
mjs	19.7/sec	28.9/sec
transicc	24.8/sec	31.5/sec
libpng	120.9/sec	157.4/sec
bmp2tiff	16.3/sec	25.7/sec
objdump	22.9/sec	27.3/sec

The results are shown in Table I. For comparison, the rate of hitting the target location was recorded to show the continuous exploration capability of the target location. It can be seen that the exploration ability of TCS to the target location has a certain improve.

IV. FUZZY TEST OPTIMIZATION METHOD BASED ON MULTI-FACTOR SEED SELECTION

In the previous chapter, the variation algorithm for target continuous exploration (TCS) was introduced, and this method can effectively improve the continuous exploration of target locations. In fact, a lot of work is needed to assist before seed variation can be performed. This chapter optimizes the fuzzy testing procedure of AFL based on TCS, which consists of the following parts: pre-processing of the procedure for the target location and multi-factor seed selection.

A. *Multi-Factorial Seed Selection Strategy*

AFL's original seed selection strategy was to select small, fast executing seeds and add them to the "favored" label, based on the code coverage to increase the scope of fuzzy exploration as much as possible to find more potential vulnerabilities, for the targeted fuzzy test can not only consider the code coverage, but also select the seeds that can reach the target location as much as possible to fully explore the target location. Therefore, the selection of seeds that can hit the target, so that they have more possibilities of variation, and thus continue to explore the target location is the basis of the seed selection strategy, and on this basis, further consideration is given to the selection and code coverage of some harder-to-hit targets.

In this section, a multi-factor seed selection strategy (MFS) is proposed to rank the input seed levels. the MFS includes:1) the number of target functions hit by the seed. 2) the number of hits at the target location. and 3) the path coverage. The execution of fuzzy tests can be easily obtained by staking them in the preprocessing stage using shared memory and thus making the selection. When performing the seed selection, it is necessary to distinguish whether the seeds hit the target location or not. For the seeds that do not hit the target location and the initial input seeds are selected using the original seed selection strategy, and if the seeds hit the target location, MFS is used for seed selection.

B. *Number of Seed Hits for the Objective Function*

The first factor to consider is the number of target functions hit by the seed. Generally, if the number of hit target locations is higher, the exploration of more target locations can be achieved in a shorter time, thus improving the efficiency of directed fuzzy testing. For complex target locations that are more difficult to hit are ignored if the consideration is whether the target location is hit or not, so considering the function where the target location is hit can preserve this information and help in the exploration of these targets.

In this section, we need to calculate the number of target functions hit by the seeds, and the distinction between whether the seeds hit the target position function can be done here. We denote this part of the weight as W_1, the total number of target positions can be obtained directly as N , and the total number of target position functions as N_f, in fact, after distinguishing whether the target function is hit or not, we can use N to calculate W_1 directly, if we want to calculate the total number of target functions will bring the extra overhead reduces the efficiency of the fuzzy test. Note that the number of target functions $N_f \leq N$, the final computed result is somewhat larger but does not affect the prioritization process. Next, the target function of the seed hit the number of hits is labeled as n_f, and the weight of the first factor can be calculated as $W_1 = n_f/N$. It can be seen that when the total number of targets is small, the number of seed hits on the target function has a large impact on W_1. Conversely, if there are many targets, a single target hit will have a small impact.

C. *Number of Hits at the Target Location*

Generally speaking, if a target location is hit many times it will be more fully explored, which often means that these target locations are easier to reach. In contrast, locations with fewer hits or even no hits are harder to find if they are potentially vulnerable. This indicates that these targets have a more complex structure or branching conditions. For these targets with fewer hits, they can be referred to as rare targets.

First, we define the concept related to rare targets. The number of executions for a target location t can be denoted as *numT [t]*, and for the input seed *s* if it hits the target location *t*, it can be denoted as *hits(s,t)*. Thus *numT [t]* can be computed using equation (1). Thus, by calculating *numT [t]*, a mapping between the seeds and the number of hits at the target position is established, which is updated each time the seeds hit the target position.

$$numT[t] = \sum_{s \in S} hits(s, t) \qquad (1)$$

After calculating the number of hits per target *numT [t]* in this way, the total number of hits for all targets can be further calculated and expressed as Num_T . Based on this we can calculate the rarity of each target position, so that if the seed hits that position, it will receive the corresponding weight. A simple idea is to calculate the proportion of target branches by this, the number of hits of the seed is inversely proportional to its weight, as in equation (3) is shown.

$$Num_T = \sum_{t \in T} numT[t] \qquad (2)$$

$$W_{2i} = Num_T/numT[t_i] \qquad (3)$$

However, if we use only this simple method, we may reduce the exploration of rare targets. Under ideal conditions, seeds that can hit rare targets take precedence over other seeds. Therefore, we must design a criterion to distinguish whether a target is a rare target or not. A natural idea is to specify a constant n and rank the target locations by the number of hits and consider them rare if their rank is lower than *n*, or when the number of hits is less than a certain percentage of the total number of input seeds.

$$numT[t] \leq rarity\_cutoff \qquad (4)$$

$$rarity\_cutoff = 2^i \text{ such that } 2^{i-1} < \min_{t \in T} (numT[t]) \qquad (5)$$

For example, if the minimum number of hits on a target location is 31, then any target location with less than 2^6 hits is considered as a rare branch. Further the rarest target location t^* can be calculated by equation (6)

$$t^* = argmin(numT[t])(t \in T) \qquad (6)$$

Based on the above rare target locations, if a seed can hit a rare target location, it will have a higher priority and will outperform other seeds that do not hit the rare target when prioritizing.

V. Experiment And Analysis Result

Based on the target-guided directed gray-box fuzzy testing approach mentioned in this paper, a directed fuzzy testing tool DTFuzz is designed and implemented based on the classical fuzzy testing tool AFL, which can perform directed fuzzy testing on software with a given source code and target location, and has a significant improvement in the exploration of target location compared with AFL and AFLGo.

A. *Experimental Setup*

We compare DTFuzz with the following fuzzy testing tools:1) AFL. the classical coverage guided gray-box fuzzy testing tool is also the tool that this tool relies on, which does not care about the target location. 2) AFLGo. is a directed gray-box fuzzy testing tool that uses a simulated annealing strategy to get as close to the target location as possible during the fuzzing process. Since both AFLGo and DTFuzz rely on AFL, we use AFL as the reference benchmark in our experiments. The selection of test cases in our experiments was based on the documentation provided by AFLGo and the Google Fuzzy Test Suite, some of which also provide CVE (Common Vulnerabilities & Exposures) id. from which we selected seven actual open source software to test and evaluate our techniques. The choice of target locations in this experiment was based on the AFLGo documentation and the crash reports from the Google test suite.

B. *Experimental Results and Evaluation*

The main focus in this experiment is on the execution speed of the loop part of the fuzzy test, and the time of the preprocessing part is not taken into account. In addition, code coverage is the classical measure of gray-box fuzzing, and although our method is directed fuzzy testing, code coverage can also reflect the ability of fuzzy testing to a certain extent. Finally, the number of crashes and unique crashes is the ultimate goal of fuzzy testing and an important ability to show the ability of fuzzy testing. Especially for DGF, the main concern is its ability to trigger a crash at the target location.

TABLE II. RESULTS OF RUNNING ON ACTUAL SOFTWARE

Actual Software	Execution speed			Code Coverage		
	AFL	AFLGo	DTFuzz	AFL	AFLGo	DTFuzz
readelf	387.3/sec	365.1/sec	385.8/sec	11597	10589	10284
xmllint	982.8/sec	851.5/sec	809.0/sec	6867	3557	5390
mjs	160.1/sec	183.4/sec	185.7/sec	3516	1881	2745
transicc	230.5/sec	166.6/sec	193.2/sec	7286	3970	5908
libpng	1543.0/sec	1168.1/sec	1063.7/sec	3467	1574	2819
bmp2tiff	162.7/sec	98.2/sec	108.4/sec	6132	3949	5574
objdump	421.7/sec	367.6/sec	418.3/sec	7186	1660	4067

Actual Software	Target hits			Unique Crash		
	AFL	AFLGo	DTFuzz	AFL	AFLGo	DTFuzz
readelf	-	44.7/sec	42.6/sec	0	0	0
xmllint	-	65.9/sec	70.6/sec	35	32	60
mjs	-	19.7/sec	25.8/sec	18	63	84
transicc	-	24.8/sec	28.6/sec	1	8	7
libpng	-	120.9/sec	140.9/sec	0	0	0
bmp2tiff	-	16.3/sec	20.1/sec	58	41	62
objdump	-	22.9/sec	31.6/sec	15	4	35

The results are shown in Table II . The execution speed of DTFuzz is not much worse than that of AFL and AFLGo on most of the software, and DTFuzz retains the original AFL fuzzy test loop structure, and the additional staking and shared memory operations are performed along the original execution path without significant overhead, which indicates that This also indicates that DTFuzz's multi-factor seed selection and variation strategies do not have a significant impact on the fuzzy test rate. It can also be seen that the execution speed of DTFuzz is slower than that of AFL, but faster than AFL on some software. Next is the code coverage, and it can be seen that DTfuzz has lower code coverage compared to the coverage guided AFL, which is an expected result since DTfuzz devotes most of its resources to exploring the target location area and accordingly has less ability to explore the entire code space. Then, the number of target hits, to better represent the ability of target hits was experimentally performed by the speed of hitting target locations, and the results show that DTFuzz can generate more inputs that can hit target locations in most cases compared to AFLGo. Finally, the calculation of the number of unique crashes shows that DTfuzz triggers more crashes than AFL and AFLGo.

TABLE III. STATISTICAL RESULTS ON THE ACTUAL SOFTWARE

Actual Software	Execution speed			Code Coverage		
	AFL	AFLGo	DTFuzz	AFL	AFLGo	DTFuzz
readelf	387.3/sec	94.3%	99.6%	11597	97.8%	97.8%
xmllint	982.8/sec	86.6%	82.3%	6867	51.8%	78.5%
mjs	160.1/sec	114.6%	116.0%	3516	53.5%	78.1%
transicc	230.5/sec	72.3%	83.8%	7286	54.5%	81.1%
libpng	1543.0/sec	75.7%	68.9%	3467	45.4%	81.3%
bmp2tiff	162.7/sec	60.4%	66.7%	6132	64.4%	90.9%
objdump	421.7/sec	87.2%	99.2%	7186	23.1%	56.6%

Actual Software	Target hits			Unique Crash		
	AFL	AFLGo	DTFuzz	AFL	AFLGo	DTFuzz
readelf	-	44.7/sec	0.95	0	0	0
xmllint	-	65.9/sec	1.07	35	0.9	1.7
mjs	-	19.7/sec	1.31	18	3.5	4.7
transicc	-	24.8/sec	1.15	1	8	7
libpng	-	120.9/sec	1.17	0	0	0
bmp2tiff	-	16.3/sec	1.23	58	0.7	1.1
objdump	-	22.9/sec	1.38	15	0.3	2.3

The relevant statistics are shown in Table III . The statistics here are based on AFLGo, except for the target location hits, which are based on AFL. Where the seed execution speed and code coverage are compared by percentages, and the number of target location hits and unique crashes are multipliers. It can be seen that DTFuzz's execution speed is about 68% of that of AFL except for bmp2tiff and libpng, where it is not too far from AFL. In terms of code coverage, except for objdump, where the coverage is low, all other software have more than 78% of the coverage of AFL. This result shows that by relying on the multi-factor seed selection strategy, it is possible to continue to detect other parts of the program to be tested after the target location is explored more fully. In terms of hitting the target location, all the software has more than 15% improvement except for readelf, which is slightly lower than AFLGo, and some software has 30% improvement above. Finally, the number of unique crashes triggered can be seen to be 1.1 times higher on bmp2tiff than on AFL, and many times higher on all other software, while the number of crashes triggered by DTFuzz can be seen to be the number of collapses is also higher than that of AFLGo, except for transicc.

The most important capability for directed fuzzy testing is the ability to continuously explore a given target location to expose crashes at the target location. We tracked the execution of these unique crashes with DTFuzz, recording the number of crashes at the target location and other locations separately, and the results are shown in Table IV. The experimental results show that most of the unique crashes are triggered at the target location. This means that our technique can be effectively used for targeted fuzzing and more crashes can be detected.

TABLE IV. RESULTS FOR THE NUMBER OF CRASHES AT THE TARGET AND OTHER LOCATIONS

Actual Software	Unique Crash	
	Other Locations	Target Location
xmllint	20	40
mjs	17	67
transicc	0	7
bmp2tiff	21	41
objdump	11	24

Based on the experimental results, it can be seen that DTFuzz triggers more crashes in xmllint, mjs and objdump than it detects in other software. As an example, xmllint, which parses one or more XML files, is useful for detecting errors in the XML code and in the XML parser itself. It has strict requirements on the input to the program, and if the input does not conform to the format, the program cannot be explored further. This result also shows that the method is able to break through complex conditional statements and keep exploring the program. It can also be seen that although AFLGo also directs the fuzzy test towards the target location and generates inputs that hit the target location, its performance is not better than DTFuzz because its mutation strategy is the same as AFL, which does not retain the previous mutation information and performs random mutation for the hit target seed.

TABLE V. EXPERIMENTAL RESULTS OF MULTI-FACTOR SEED SELECTION STRATEGY AND DIRECTED VARIATION STRATEGY, RESPECTIVELY

Actual Software	**Unique Crash**		**Code Coverage**	
	DTFuzz-s	**DTFuzz-m**	**DTFuzz-s**	**DTFuzz-m**
xmllint	23	52	6715	3783
mjs	18	47	3142	1421
transicc	5	10	5347	3946
bmp2tiff	37	32	6347	2742
objdump	18	26	6871	1293

In addition, further analysis of the effectiveness of the TCS and MFS proposed in this paper is needed. We run DTfuzz with only the multi-factor seed selection strategy (DTFuzz-s) and DTfuzz with only the directed variation strategy (DTFuzz-m) on the software that triggers a crash, where only the software that triggers a crash is considered, and the results are shown in Table V. Based on the experimental results, we can see that DTFuzz-m can trigger more crashes. This is because the mutation strategy ensures that the seeds that can reach the target location are continuously generated in subsequent mutations, generating more seeds that hit the target location and thus triggering more crashes. The number of crashes triggered by DTFuzz-s is similar to that of AFL. This is because for the seed selection strategy, although seeds that are more likely to reach the target location can be selected, giving them more mutation possibilities, there is no guarantee that the seeds generated by mutation can still hit the target location, resulting in fewer crashes triggered. Similarly, without a seed selection strategy, DTFuzz cannot fully utilize the target location information and may waste a lot of resources on irrelevant seeds or repeatedly execute seeds that are more likely to hit the target location, resulting in a decrease in the ability of fuzzy testing.

By comparing DTFuzz-m without the seed selection strategy with DTFuzz, it can be seen that the use of the seed selection strategy on top of the directed variation algorithm improves the coverage of the code by the fuzzy tests and improves the shortcomings of the variation algorithm. This is also in line with our expectation that after the "easier" targets are fully explored, the priority of the seeds that hit these target locations will be lowered to better explore other targets. In particular, the rare targets that are "harder" to hit will be explored continuously, thus improving the ability of the targeted ambiguity test.

VI. CONCLUSION

The goal-oriented directed gray-box fuzzy testing technique based on the randomness and blindness of variation in gray-box fuzzy testing is proposed, which mainly includes the directed variation strategy of goal continuous exploration. It identifies the key nodes linking the input to the target location in the program to be tested by traversing the input seeds, and then uses the directed variation algorithm to handle the deterministic variation phase and the random variation phase to guide the fuzzy test toward the target location, respectively. At the same time, the gray-box fuzzy test is further optimized based on the directed variation strategy by using the LLVM framework to stake the target location information into the program to be tested based on the original staking code, and then using a multi-factor seed selection strategy to select the seeds that are more likely to hit the target. The concept of rare target is proposed to reduce the number of "useless" duplicate seeds generated for the "easier" hit targets, and give higher priority to the harder hit target locations, which is a good way to improve the target location exploration under complex conditions for the directed fuzzy test. The exploration also improves the path coverage.

The method still has the following problems: 1) the target location is still mainly dependent on manual analysis of the program to be measured for the static analysis tool to provide the analysis report due to the number of generally large can not be well used.2) For how to generate the seeds to reach the target location mainly relies on the random variation process, and no suitable scheme is given to generate the seeds to reach the target location directly.

REFERENCES

[1] *Miller B P, Fredriksen L, So B. An Empirical Study of the Reliability of UNIX Utilities [J/OL]. Commun. ACM. 1990, 33 (12): 32-44. https://doi.org/10.1145/96267. 96279.*

[2] *Böhme M, Pham V-T, Roychoudhury A. CoverageBased Greybox Fuzzing as Markov Chain [C/OL]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York,NY,USA,2016:1032-1043. https://doi.org/10.1145/2976749.2978428.*

[3] *Zhao L, Duan Y, Yin H, et al. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing [C]. Network and Distributed System Security Symposium. 2019.*

[4] Chen C, Cui B, Ma J, et al. A systematic review of fuzzing techniques [J]. Computers Security. 2018: 118–137.

[5] *American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.*

[6] *Li Y, Ji S, Lv C, et al. V-fuzz: Vulnerability-oriented evolutionary fuzzing [J]. arXiv preprint arXiv:1901.01142. 2019.*

[7] *You W, Zong P, Chen K, et al. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits [C]. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2139–2154.*

[8] *Böhme M, Pham V-T, Nguyen M-D, et al. Directed greybox fuzzing [C]. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329–2344.*

[9] *Zong P, Lv T, Wang D, et al. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning [C]. 29th {USENIX} Security Symposium ({USENIX} Security 20). 2020: 2255–2269.*

[10] *Ye J, Li R, Zhang B. RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation [J]. Mathematical Problems in Engineering. 2020, 2020.*