Zhang JM, Cui ZQ, Chen X *et al.* DeltaFuzz: Historical version information guided fuzz testing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(1): 29–49 Jan. 2022. DOI 10.1007/s11390-021-1663-7

DeltaFuzz: Historical Version Information Guided Fuzz Testing

Jia-Ming Zhang[1] (张家铭), Zhan-Qi Cui[1,*] (崔展齐), *Senior Member*, *CCF*, *Member*, *IEEE*
Xiang Chen[2] (陈 翔), *Senior Member*, *CCF*, *Member*, *IEEE*, Huan-Huan Wu[1] (吴欢欢)
Li-Wei Zheng[1] (郑丽伟), *Member*, *CCF*, and Jian-Bin Liu[1] (刘建宾)

[1] *Computer School, Beijing Information Science and Technology University, Beijing 100101, China*
[2] *School of Information Science and Technology, Nantong University, Nantong 226019, China*

E-mail: {radon, czq}@bistu.edu.cn; xchencs@ntu.edu.cn; {wuhuanhuan, zlw, ljb}@bistu.edu.cn

Received June 1, 2021; accepted December 16, 2021.

Abstract With the widespread use of agile software development methods, such as agile and scrum, software is iteratively updated more frequently. To ensure the quality of the software, regression testing is conducted before new versions are released. Moreover, to improve the efficiency of regression testing, testing efforts should be concentrated on the modified and impacted parts of a program. However, the costs of manually constructing new test cases for the modified and impacted parts are relatively expensive. Fuzz testing is an effective method for generating test data automatically, but it is usually devoted to achieving higher code coverage, which makes fuzz testing unsuitable for direct regression testing scenarios. For this reason, we propose a fuzz testing method based on the guidance of historical version information. First, the differences between the program being tested and the last version are analyzed, and the results of the analysis are used to locate change points. Second, change impact analysis is performed to find the corresponding impacted basic blocks. Finally, the fitness values of test cases are calculated according to the execution traces, and new test cases are generated iteratively by the genetic algorithm. Based on the proposed method, we implement a prototype tool DeltaFuzz and conduct experiments on six open-source projects. Compared with the fuzzing tool AFLGo, AFLFast and AFL, DeltaFuzz can reach the target faster, and the time taken by DeltaFuzz was reduced by 20.59%, 30.05% and 32.61%, respectively.

Keywords fuzz testing, regression testing, change impact analysis, fitness function

1 Introduction

Software maintenance is a particularly important stage in the software life cycle, and its purpose is to fix software vulnerabilities and ensure that the software can run stably [1, 2]. Currently, with the widespread use of agile software development methods and mobile Internet, software is updated more frequently. For example, the first version of mobile app Facebook for iOS was released in 2008①. Since 2015, the app is updated nearly every week. By April 17, 2021, it had been updated more than 300 times, and the latest version was 314.1②. In the software maintenance and update process, vulnerabilities will be detected and repaired, new functions will be introduced, and a new version of the software program will be produced. Comparing the new version with the last version of software, many parts of the software are impacted by the updates, and these updates are likely to cause unknown vulnerabilities. The locations where the software source code updates during software evolution are called change points in this paper. Regression testing will be performed before the new version of software is released to prevent vulnerabilities from impacting the use of the software. However, due to the high update frequency of software,

Regular Paper
Special Section on Software Systems 2021—Theme: Dependable Software Engineering
This work was partially supported by the Leading-Edge Technology Program of Jiangsu Natural Science Foundation of China under Grant No. BK20202001, the National Natural Science Foundation of China under Grant No. 61702041, and the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project under Grant No. QXTCP C201906.
*Corresponding Author
①https://www.adweek.com/performance-marketing/facebook-for-iphone-application-launches/, Apr. 2021.
②https://apps.apple.com/us/app/facebook/id284882215, Apr. 2021.

frequent regression tests are required, which require considerable manpower and computing resources. If regression testing is blindly performed on the new version of a program, the test of the change points will be insufficient, and many resources will be spent unnecessary, even delaying the release of the new version [3]. Therefore, appropriate test cases need to be filtered out from the existing test case repository for reuse and adaptation, and new test cases also need to be constructed before regression testing. If test cases covering the change points can be automatically generated, the efficiency of regression testing will be improved.

Fuzz testing, which was first proposed by Miller *et al.* in 1989 [4], can automatically generate a large number of new test cases and send them to the program being tested [5–7]. For example, AFL③ is a commonly-used mutation-based fuzzing tool that exposes hundreds of high-risk vulnerabilities, and many studies attempt to improve the efficiency of fuzzing based on AFL [8–11]. According to different extents of internal logic analysis, fuzz testing can be divided into black-box fuzz testing [12], white-box fuzz testing [13, 14] and grey-box fuzz testing [15–19]. Among them, grey-box fuzz testing has been one of the most effective techniques for detecting vulnerabilities in recent years. Grey-box fuzzing can use lightweight program analysis with a small overhead [16, 19] to achieve excellent test results. Although fuzz testing has many advantages, its shortcomings cannot be ignored. For example, coverage-guided fuzz testing tools such as AFL aim at a higher code coverage, which makes these tools difficult to use for regression testing. Target-guided fuzz testing tools such as AFLGo [20] aim to cover one or more code blocks without considering the paths impacted by the change points.

To solve the above problems, we propose a method that uses historical version information guided fuzz testing. First, the program being tested and its last version are compared to acquire basic blocks where the change points are located. Second, change impact analysis is performed to generate a set of impacted basic blocks and the program being tested is instrumented according to the impacted basic block set. Finally, fuzz testing is performed for the program being tested and outputs a testing report when it is aborted. During the testing process, the fitness value of a test case is calculated according to the path it covers, and corresponding test resources are allocated to the test case. We implement a prototype tool DeltaFuzz based on the above method and conduct experiments on six open-source projects. The experimental result shows that DeltaFuzz outperforms AFLGo in terms of the number of triggered crashes, detected vulnerabilities, and covered paths.

The main contributions of this paper can be summarized as follows.

• A fuzz testing method guided by historical version information is proposed. This method uses the differences between the program being tested and its last version to generate the set of basic blocks impacted by change points. The impacted basic blocks set provides guidance for the fuzzing tool to improve the testing efficiency.

• A path-sensitive fitness function is proposed. This function calculates the fitness value based on the impacted basic block set and the path covered by the test case. The fitness function can help fuzz testing allocate resources effectively.

• Based on the above method, the prototype tool DeltaFuzz is implemented, and experiments are conducted on six open-source projects to validate the effectiveness of the proposed method.

2 Historical Version Information Guided Fuzz Testing

The framework of the historical version information guided fuzz testing method proposed in this paper is shown in Fig. 1. First, the source files of the program being tested and the corresponding historical version program are acquired, and the change points in the program being tested are located by comparing the differences. Then, change impact analysis is performed according to the locations of the change points, and the source files of the program being tested are instrumented according to the analysis result. Finally, fuzz testing is performed on the program being tested after instrumentation, and test resources are allocated to test seeds according to the fitness. The testing process is aborted when the abort condition is satisfied.

2.1 Change Points

In the software evolution process, developers update the old version of software by adding branches, modifying parameter types, and introducing new functions and other actions. These change points are error-prone and will change the behaviors of the software.

③https://github.com/google/AFL, Jan. 2021.

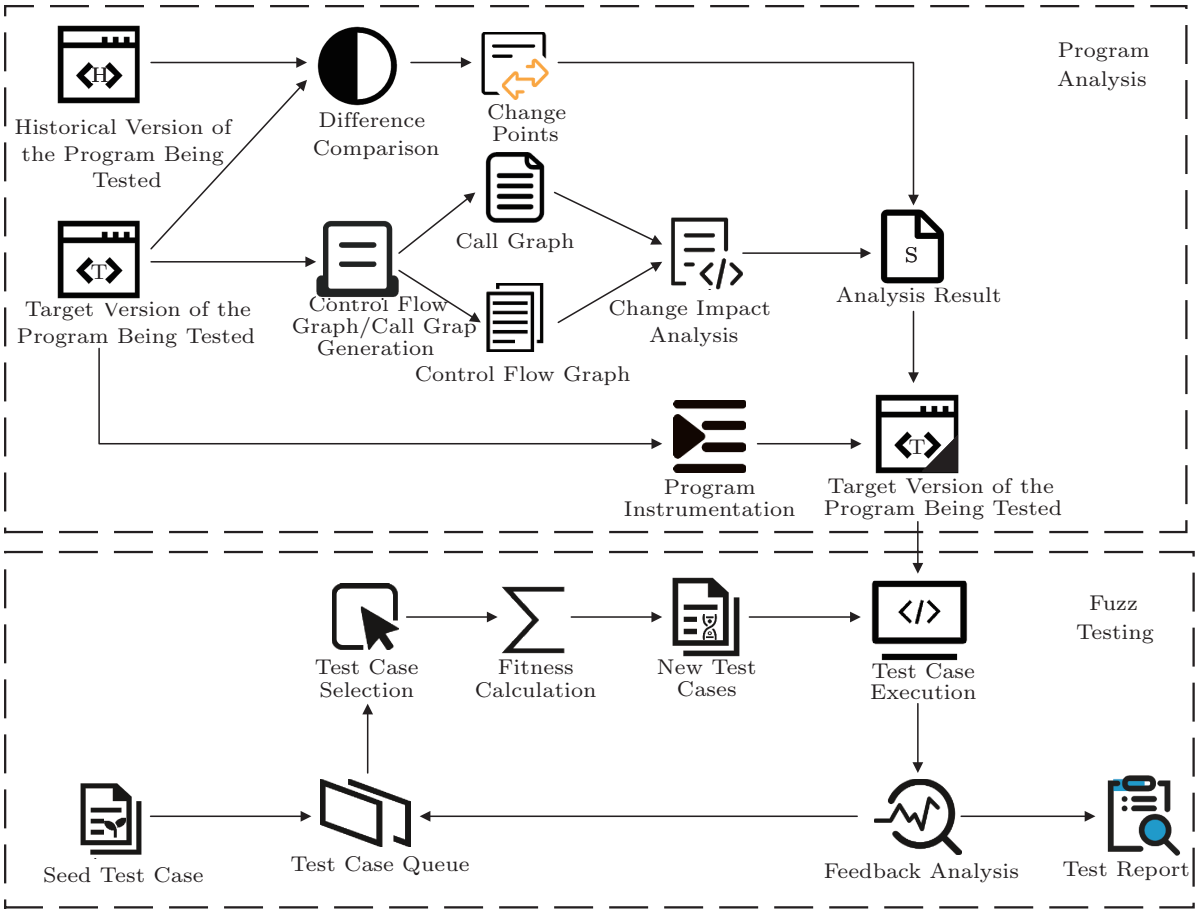


Fig.1. Framework of historical version information guided fuzz testing.

For example, Fig. 2(a) is a code snippet from the listfdb.c file in version 0.4.8 of the libming project④, and Fig. 2(b) is a code snippet from the listfdb.c file in version 0.4.7 of the libming project⑤. In these two code snippets, we can see that there is a code change in the readBits function (in red). In line 74 of version 0.4.7 of the libming project, the code is "++fileOffset". In version 0.4.8 of the libming project, the code is changed to "if(feof(f))". Therefore, compared with version 0.4.7 of the libming project, line 74 of the listfdb.c file in version 0.4.8 of the libming project is a change point.

2.2 Path-Sensitive Suspicious Analysis for Basic Blocks

Modifications, improvements, or additions of features in the new version of software will introduce change points. In the version difference analysis process, we need to locate the change points to perform change impact path analysis. However, even a simple program may have numerous paths. Large overhead will be introduced if change impact analysis is performed on all paths of a program. Therefore, we propose performing path-sensitive suspicious analysis for basic blocks, which are sequences of statements executed sequentially to replace the change impact analysis on paths. In this way, we can obtain a set of basic blocks that are impacted by the change points.

To determine whether a basic block is impacted by any change points, reachability analysis can be performed between it and the change points. Specifically, there are two reachability analysis methods to determine whether a basic block is impacted by the change points. One method is forward analysis, which starts from the basic block being analyzed and takes a basic block that contains change points as the target to traverse the control flow graph. If the target can be reached by the basic block being analyzed, it indicates that the basic block being analyzed is impacted. For-

④https://github.com/libming/libming/tree/ming-0_4_8, Feb. 2021.
⑤https://github.com/libming/libming/tree/ming-0_4_7, Feb. 2021.

Code Snippet 1

```
55 int readBits(FILE *f, int number)
56 {
57     int ret = buffer;
       ⋮
66     if(number > bufbits)
67     {
68         number −= bufbits;
69
70         while(number > 8)
71         {
72             ret <<= 8;
73             ret += fgetc(f);
74             if (feof(f))
               ⋮
```

(a)

Code Snippet 2

```
55 int readBits(FILE *f, int number)
56 {
57     int ret = buffer;
       ⋮
66     if(number > bufbits)
67     {
68         number −= bufbits;
69
70         while(number > 8)
71         {
72             ret <<= 8;
73             ret += fgetc(f);
74             ++fileOffset;
               ⋮
```

(b)

Fig.2. Code snippets in listfdb.c of the libming project. (a) Code snippet in version 0.4.8. (b) Code snippet in version 0.4.7.

ward analysis is iterated until the entry point of the program is reached. The other method is backward analysis, which takes the basic block being analyzed as the end point and takes a basic block that contains change points as the start point to traverse the control flow graph in a backward manner. If the end point can be reached by the start point, it indicates that the basic block being analyzed is impacted. Due to the existence of loops in a program, too much backward analysis may cause all the basic blocks in the program to be impacted by the change points. Therefore, the backward analysis only proceeds to the end of the current function. If the basic block being analyzed and the basic blocks that contain change points cannot be reached in the above two ways, it means that the basic block being analyzed is not impacted. Through this method, we can determine whether a basic block in the path segment is impacted before fuzz testing. After fuzz testing starts, the fitness of a test case is calculated based on whether the basic blocks covered by the test case are impacted.

The code snippet in Fig.2(a) is adapted from version 0.4.8 of the libming project to illustrate the reachability analysis process of the basic block. To facilitate the subsequent description, we use [bbName] to represent the basic block bbName and <FuncName> to represent the function FuncName. As shown in Fig.2, the historical version of libming (version 0.4.7) is compared with the target version (0.4.8), and one change point is identified in line 74 of the listfdb.c file. The control flow graph of the code snippet in Fig. 2(a) is shown in Fig.3 and is generated by the graph extractor of AFLGo. In Fig.3, the pink basic block represents the basic block where the change point is located, namely, [listfdb.c:72]. Taking the pink basic block as the start or end, we search for basic blocks that can reach the pink basic block using forward or backward analysis. Once a basic block can be reached in the forward or backward direction, it means that the basic block is impacted by [listfdb.c:72]. Fig.3 shows that except for the basic blocks [listfdb.c:104] and [listfdb.c:61], all the other basic blocks are impacted by [listfdb.c:72]. Since the basic block [listfdb.c:72] is in the function <readBits>, the basic block that calls the function <readBits> needs to be analyzed next after finishing the analysis of the control flow graph of the <readBits> function.

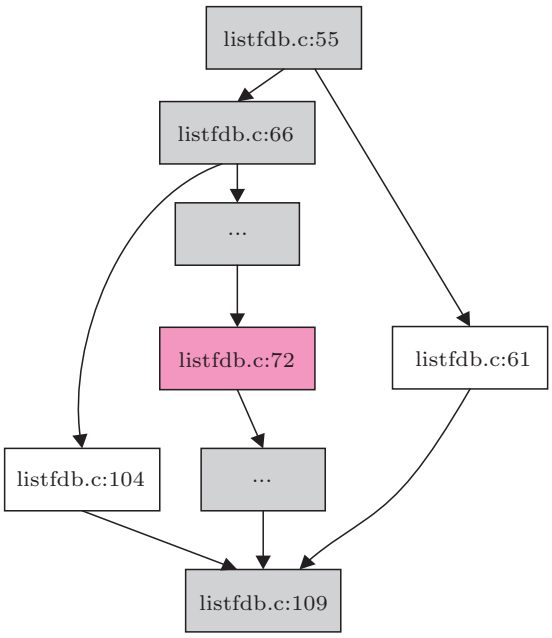


Fig.3. Part of the control flow graph of the <readBits> function.

Traversing the function call information, we find that the basic block [parser.c:101], which is in the <parseSWF_MATRIX> function, calls the <readBits> function. The control flow graph of the <parseSWF_MATRIX> function is shown in Fig.4(a). In Fig.4, the orange basic block means that it can reach the change point through function calls. For example, the basic block [parser.c:101] calls the <readBits> function, which contains the change point, marked in orange. The reachability analysis method mentioned above can be used iteratively to analyze whether the basic blocks in <parseSWF_MATRIX> are impacted. Therefore, Fig.4(a) shows that the basic blocks are all impacted by the change point.

After analyzing the control flow graph of the <parseSWF_MATRIX> function, the basic blocks that call the function need to be located. The <parseSWF_MATRIX> function is called by the basic block [parser.c:591], which is contained in the <parseSWF_FILLSTYLE> function. The <parseSWF_FILLSTYLE> function is called by the basic block [parser.c:655], which is contained in the <parseSWF_LINESTYLE2> function. The part of the control flow graph of the <parseSWF_FILLSTYLE> function and the complete control flow graph of the <parseSWF_LINESTYLE2> function are shown in Fig.4(b) and Fig.4(c), respectively.

Algorithm 1 introduces the path-sensitive suspicious analysis method for basic blocks. The input is a set of change points acquired through the difference comparison and stored in the queue *bbQueue*. The output is *bbSusp*, which stores the basic blocks that are impacted by the change points. In the analysis process, first, when *bbQueue* is not empty, the analysis will be performed continuously (line 1). Then, the first element of *bbQueue* will be obtained (line 2), and it will be set as the basic block being analyzed bb_T (line 3). Variable *bbChecked* is introduced to prevent the path set analysis from falling into an infinite loop. For example, let us suppose there is a case where <FuncC>→<FuncA>→<FuncB>→<FuncC>→ the function of the change point. If we do not add the checked basic blocks to *bbChecked*, the program will infinitely fall into the loop of analyzing <FuncC>→ analyzing <FuncB>→ analyzing <FuncA>→ analyzing <FuncC>→ .... This will cause the analysis to fail to end. *bbChecked* is set to prevent this case. If bb_T has been analyzed, that is, it has been already in the *bbChecked* queue, it will be skipped, and the next basic block in the queue will be analyzed. The control flow graph where bb_T is located and the function where bb_T is located will be stored in CFG and $Func$ (lines 6 and 7). Next, we analyze whether the other basic blocks in CFG are impacted (lines 8–14). The impacted basic block will be added to the set *bbSusp* (lines 9–12). Finally, other basic blocks that call the function $Func$ will be added to be analyzed queue *bbQueue* (line 15), and bb_T is added to the analyzed basic block queue *bbChecked* to prevent an infinite loop (line 16).

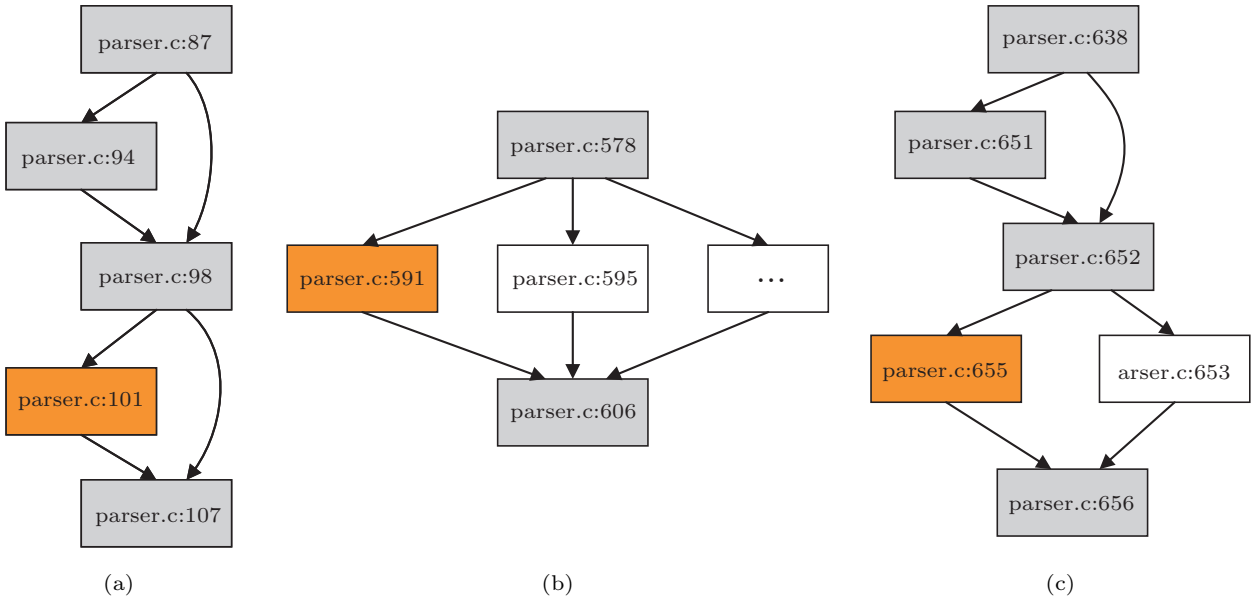


Fig.4. Control flow graphs of the three functions. (a) <parseSWF_MATRIX> function. (b) Part of the <parseSWF_FILLSTYLE> function. (c) <parseSWF_LINESTYLE2> function.

Algorithm 1. Path-Sensitive Suspicious Analysis for Basic Blocks

Input: queue *bbQueue*: a queue of change points
Output: set *bbSusp*: a set of impacted basic blocks

```
1: while bbQueue is not empty do
       //bb_T is the basic block under analysis
2:     bb_T := bbQueue.pop()
3:     if bb_T in bbChecked then
4:         Continue
5:     end if
6:     CFG := the control flow graph of the function which contains bb_T
7:     Func := the function where bb_T is located
8:     for bb in CFG do
9:         if bb and bb_T are the same basic block then
10:            Add bb to bbSusp
11:        else if bb can reach bb_T or bb_T can reach bb then
12:            Add bb to bbSusp
13:        end if
14:    end for
15:    bbQueue.push(basic blocks which called Func)
16:    bbChecked.push(bb_T) //Add bb_T to bbChecked
17: end while
18: return bbSusp
```

2.3 Test Case Fitness Analysis

The execution path of t_i is defined as a sequence of basic blocks $P_i = <bb_0, bb_1, ..., bb_n>$, which is covered by test case t_i. Among them, bb_0 is an entry point of the program, and bb_n is one of the exit points of the program. $<bb_j, bb_{j+1}>$ $(0 \leqslant j < n)$ is the basic segment that constitutes the execution path P_i. Sequence $PS^i_{p,q} = <bb_p, bb_{p+1}, ..., bb_q>$ $(0 \leqslant p < n$ and $p < q \leqslant n)$ is a path segment of P_i.

Fig.5 is used to illustrate the above definition. In Fig.5, $P_i = <bb_0, bb_1, ..., bb_n>$ is a path of the program, which is executed by test case t_i. bb_0 is the entry point of the program, and bb_n is one of the exit points of the program. $PS^i_{0,3} = <bb_0, bb_1, bb_2, bb_3>$ is a path segment of P_i and $bs^i_{2,3} = <bb_2, bb_3>$ is a basic segment of P_i.

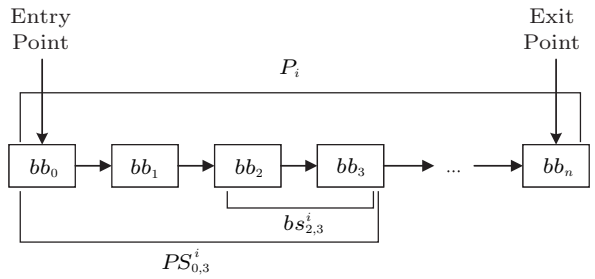


Fig.5. Example of program path, path segment and basic segment.

We introduce (1) to calculate whether a basic segment is suspicious, and $bs^i_{x,x+1}$ represents a basic segment in the path. During the execution of the test case, the first basic block bb_x and the second basic block bb_{x+1} are examined to determine whether they are included in the suspicious basic block set *bbSusp*, that is, whether they are impacted by change points. If both bb_x and bb_{x+1} are in *bbSusp*, which means that they are impacted by change points, then $bs^i_{x,x+1}$ is a suspicious basic segment. If one or both of bb_x and bb_{x+1} are not in *bbSusp*, which means that one or both of them are not impacted by the change points, then $bs^i_{x,x+1}$ is not a suspicious basic segment.

Because the fitness values of test cases are evaluated in a path-sensitive manner, the proportion of paths that are impacted by the change points in the test case execution trace must be considered. The definition of the basic segment shows that the basic segment is combined with two consequent basic blocks, which are connected by a path.

In forward analysis, if the starting basic block and the ending basic block of a basic segment are both impacted by a change point, it means that the test case may reach the change point after covering the basic segment. Therefore, it can be considered that this basic segment is impacted by the change point. Conversely, if only 1 or 0 basic block in the basic segment is impacted by a change point, it means that the test case cannot reach the change point along with the basic segment, thereby the basic segment is not impacted by the change point.

In backward analysis, if the starting basic block and the ending basic block of a basic segment are impacted by a change point, it means that the basic segment can be covered by paths that can cover the change point. In this case, the basic segment has a strong correlation with the change point, thereby it can be considered that the basic segment is impacted by the change point. Conversely, if only 1 or 0 basic block in the basic segment is impacted by a change point, it means that the basic segment cannot be covered by any test case that can cover the change point, thereby the basic segment is not impacted by the change point.

$$Susp(bs^i_{x,x+1}) = \begin{cases} 1, & \text{if } bb_x \in bbSusp \text{ and } bb_{x+1} \in bbSusp, \\ 0, & \text{if } bb_x \notin bbSusp \text{ or } bb_{x+1} \notin bbSusp. \end{cases} \quad (1)$$

In order to allocate resources for test cases effectively, the fitness of test cases is calculated based on the coverage of path segments. If the path-sensitive suspicious analysis for basic blocks is performed during

the testing process, unnecessary overhead will be introduced. To improve the efficiency of fuzz testing, suspicious analysis should be completed before fuzz testing starts. Then, the program being tested is instrumented according to the suspicious analysis results. Instrumentation can help to check whether a covered basic block is impacted by change points during test case execution. If one basic block is not in the set *bbSusp*, it means that the basic block is not impacted by the change points. If one basic block is in the set *bbSusp*, it means that the basic block is impacted by the change points. According to whether the basic blocks are included in the set *bbSusp* and (1), the number of suspicious basic segments covered by the test case can be obtained to calculate the fitness of the test case.

The fitness function $Fit(t_i)$ of the test case t_i is shown in (2), and the formula is used to determine the amount of test resources allocated to t_i. P_i is the path executed by t_i. $PS^i_{u,v}$ is a segment of path P_i that consists of the continuous basic blocks acquired by change impact analysis, and these basic blocks are closely related to the change point. According to the relationship between the basic block in the path segment and change points, the basic blocks in the path segment can be divided into three types: the basic block is not impacted by change points, but it is in the same function that contains other impacted basic blocks; the basic block is impacted by change points, but it does not contain any change point; and the basic block is impacted by change points, and it contains one or more change points. $PS^i_{u,v}$ in the denominator represents the path segment of test case t_i. $|PS^i_{u,v}|$ represents the length of the path segment in the execution path of test case t_i and, more specifically, is the number of basic segments in the recorded path segment. $Susp(bs^i_{x,x+1})$ is used to analyze whether basic segment $bs^i_{x,x+1}$ is suspicious, that is, whether $bs^i_{x,x+1}$ is impacted by change points. $\Sigma_{bs^i \in PS^i_{u,v}} Susp(bs^i_{x,x+1})$ is the number of suspicious basic segments in $PS^i_{u,v}$. By using (2), the fitness of the test case t_i can be calculated. The greater the proportion of suspicious basic segments in $PS^i_{u,v}$, the larger the value of $Fit(t_i)$, and test cases with larger values should be given more test resources because these test cases are more likely to generate new test cases that can cover the change points. The lower the proportion of suspicious basic segments in $PS^i_{u,v}$, the smaller the value of $Fit(t_i)$, and test cases with smaller values should be given fewer test resources because these test cases are unlikely to generate new test cases that can cover the change points.

$$Fit(t_i) = \frac{\Sigma_{bs^i_{x,x+1} \in PS^i_{u,v}} Susp(bs^i_{x,x+1})}{|PS^i_{u,v}|}. \tag{2}$$

For example, the paths covered by test cases t_r, t_g and t_b are marked in red, green, and blue, respectively, in Fig.6. H, marked in pink, is a basic block where the change point is located. Fig.6 shows that t_r covers three basic segments: $bs^r_{A,B}, bs^r_{B,E}$ and $bs^r_{E,H}$. These three basic segments are all suspicious basic segments because A, B, E, and H are impacted basic blocks. Therefore, the fitness value of the test case t_r is $Fit(t_r) = 3/3 = 1.0$. Test case t_b covers two basic segments: $bs^b_{A,B}$ and $bs^b_{B,D}$. Among these two segments, $bs^b_{A,B}$ is a suspicious basic segment because A and B are impacted basic blocks, and $bs^b_{B,D}$ is not a suspicious basic segment because D is not an impacted basic block. Therefore, the fitness value of test case t_b is $Fit(t_b) = 1/2 = 0.5$. Test case t_g covers two basic segments: $bs^g_{A,C}$ and $bs^g_{C,G}$. Neither of them is a suspicious basic segment because C and G are not impacted basic blocks. Therefore, the fitness value of the test case t_g is $Fit(t_g) = 0/2 = 0$.

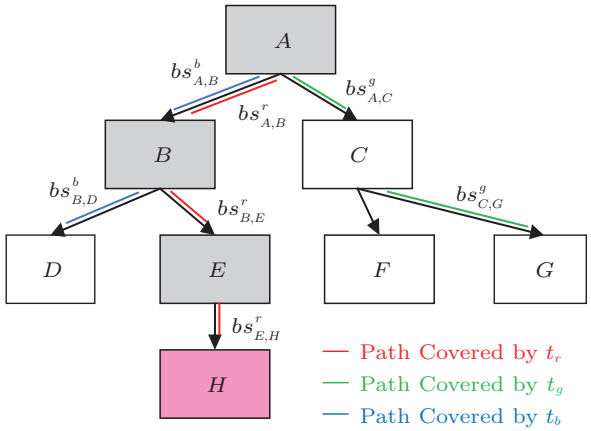


Fig.6. Example of control flow graph with test case execution path segments.

2.4 Path-Sensitive Grey-Box Fuzz Testing

Fuzzing is a testing technique that sends a large number of randomly generated test cases to the program being tested, and monitors the status of the program [21–23]. According to different extents of source code analysis, fuzz testing can be divided into black-box fuzz testing, grey-box fuzz testing and white-box fuzz testing.

We use the path-sensitive grey-box fuzz testing method, which aims to concentrate more testing resources on the basic blocks impacted by the change points. First, the code between different versions is analyzed, and the basic block that contains change points is located. Second, change impact analysis is performed to acquire the set of impacted basic blocks for instrumentation. Finally, fuzz testing is performed. During fuzz testing, the fitness is calculated according to the number of suspicious basic segments and the length of the path segment, and resources are allocated to the test cases.

Algorithm 2 describes the process of path-sensitive grey-box fuzzing. We extend the directed grey-box fuzzing approach AFLGo, and the lines with grey backgrounds represent the part we have modified. The input S is a set of initial test seeds. When the abort condition is not satisfied (line 1), the fuzzing tool will select a seed from the set and obtain its execution path information (line 2). Then, the fitness of the seed is calculated based on the length of the path segment covered by the seed and the number of suspicious basic segments. There is a scoring function in AFL, which will score the test case based on the execution time of the test case, the foundation time of the test case, the size of the test case, the path depth, and other factors. The higher the score is, the more the mutation opportunities there will be for the test case. To allocate resources to each test case more reasonably and make fuzzing more efficient, we integrate the scoring function of AFL with the fitness function as $p = p_{\mathrm{AFL}}(s) \times (Fit(s) + c)$ (line 3). Among the variables, p is the final score of test case s; $p_{\mathrm{AFL}}(s)$ is the score calculated by AFL for test case s; $Fit(s)$ is the fitness of s, which is calculated by the fitness function; and c is a constant, which is used to prevent the value of p from reaching 0 when the value of fitness is 0 and will cause the test case to have no opportunity to be mutated. The larger the value p, the greater the amount of resources that are allocated to the seed (line 4). Based on the allocated resources, seed s is mutated to generate a new test case s' (line 5). If the mutated test case s' triggers a program crash (line 6), s' is saved in S_{crash}, which is a set of test cases that can trigger crashes (line 7). Then, whether s' can increase the code coverage is judged in line 8; and if the coverage is increased, s' is added to the seed set S (line 9). When the abort condition is satisfied, the fuzz testing is aborted, and the test report is outputted (line 13).

Algorithm 2. Path-Sensitive Grey-Box Fuzzing

Input: seed set S
Output: test report $TestReport$

```
    //the abort condition can be the amount of testing time, the
    number of crashes, and so on
1:  while abort condition not satisfied do
        //Choose a seed from the set
2:      s = ChooseOneSeed(S)
        //Calculate the fitness of the seed and integrate fitness
    with scoring function of AFL
3:      p = p_AFL(s) × (Fit(s) + c)
        //Allocate resources based on the value of p
4:      for i from 1 to p do
5:          s' = Mutate(s)
6:          if s' triggers program crash then
7:              Add s' to S_crash
8:          else if IsInteresting(s') then
9:              Add s' to S //If the test case covers the new code,
                save it
10:         end if
11:     end for
12: end while
13: return TestReport
```

3 Experimental Evaluation

Based on the above methods, we implement a historical version of the information-guided fuzz testing tool DeltaFuzz on top of the target-guided grey-box fuzzing tool AFLGo. AFLGo[⑥] is a directed grey-box fuzzing tool that is implemented based on the grey-box fuzzing tool AFL. AFLGo adds the target guidance function while maintaining the efficiency of grey-box fuzzing, and a simulated annealing algorithm is used to guide the test case generation. AFLFast[⑦] is another grey-box fuzzing tool that is implemented based on AFL. AFLFast uses the Markov chain model to determine the testing resources allocated to the seed test cases. Since AFLGo, AFLFast and AFL have been widely used [24–28], they are chosen as baseline tools for comparison. Experiments are conducted on six open-source projects to evaluate the effectiveness of DeltaFuzz. In the experiments, we aim to answer the following research questions.

*RQ*1. What are the performance differences among DeltaFuzz, AFLGo and AFLFast in terms of executing test cases, triggering crashes, and covering paths?

⑥https://github.com/aflgo/aflgo, Jan. 2021.
⑦https://github.com/mboehme/aflfast, Apr. 2021.

*RQ*2. In comparison with AFLGo, AFLFast and AFL, can DeltaFuzz concentrate more resources on testing change points?

*RQ*3. In comparison with AFLGo and AFLFast, can DeltaFuzz find more vulnerabilities?

The purpose of RQ1 is to examine the general testing effect of DeltaFuzz, including the number of covered paths, executed test cases, and triggered crashes. The purpose of RQ2 is to examine the validity of DeltaFuzz, that is, compared with AFLGo and AFLFast, whether it can cover more basic blocks which contain change points, and compared with AFLGo, AFLFast and AFL whether it can reach the target basic blocks faster. The purpose of RQ3 is to examine the vulnerabilities detection ability of DeltaFuzz, that is, the difference of vulnerabilities detected by AFLGo, AFLFast and DeltaFuzz.

3.1 Experimental Design

To answer these research questions, a set of experiments are designed as follows.

The statistics of the experimental projects are shown in Table 1. In Table 1, six open-source software programs are selected out of the 11 experimental projects. They are evaluated by AFLGo [20] and are taken as the experimental projects. The target version is the same as the version of the program being tested that can be found in the fuzzing script provided by AFLGo. AFLGo provides fuzz testing scripts, which include the commit ID of the program being tested, and the corresponding version of the program being tested can be found according to the commit ID[8]. The historical version is the last version published before the version of the program being tested. The six open-source software programs are giflib[9], jasper[10], libming[11], libxml2[12], lrzip[13] and mjs[14]. The other five projects are not selected since no historical version can be found. Giflib is a software used to generate and decode gif format images, and the development language is C language. Jasper is an image processing software that can process image data in various formats. Libming is an output library that can be used in C, C++, and Java; and it contains many Flash functions. Libxml2 is the XML C parser and toolkit that was developed by Gnome and possesses high portability. It can be built and run on various systems. Lrzip is a compression program that can achieve a higher compression rate and a faster compression speed when compressing larger files. Mjs is an embedded JavaScript engine for C or C++ specifically designed for microcontrollers with limited resources. The column target version in Table 1 is the version of the program being tested, the column historical version is the last version of the program being tested, and the column lines of code is the size of the program being tested.

In the experiment, call graphs and control flow graphs are generated by LLVM[15], which is the same as the graph extractor of AFLGo. The change points are set as targets for AFLGo and DeltaFuzz. The relevant information of the change points is shown in Table 1. In the giflib project, there are 2 642 basic blocks. Among them, 59 basic blocks contain change

Table 1. Statistics of Experimental Objects

ID	Project	Target Version	Historical Version	Lines of Code	Number of Basic Blocks Containing Change Points	Number of Impacted Basic Blocks	Number of Basic Blocks
1	giflib	5.1.1	5.1.0	44 528	59	698	2 642
2	jasper	1.900.3	1.900.2	60 045	808	7 112	9 467
3	libming	0.4.8	0.4.7	128 979	239	1 520	11 409
4	libxml2	2.9.3	2.9.2-rc2	529 541	130	36 546	75 719
5	lrzip	0.640	0.631	54 481	245	2 136	6 269
6	mjs	2.2	2.1	36 468	128	1 818	3 173

[8] https://github.com/aflgo/aflgo/tree/master/scripts, Jan. 2021.
[9] http://giflib.sourceforge.net/, Feb. 2021.
[10] https://jasper-software.github.io/jasper/, Feb. 2021.
[11] http://www.libming.org/, Feb. 2021.
[12] http://www.xmlsoft.org/, Feb. 2021.
[13] https://github.com/ckolivas/lrzip, Feb. 2021.
[14] https://github.com/cesanta/mjs, Feb. 2021.
[15] https://releases.llvm.org/, Feb. 2021.

points, and 698 basic blocks are impacted. In the jasper project, there are 9 467 basic blocks. Among them, 808 basic blocks contain change points, and 7 112 basic blocks are impacted. In the libming project, there are 11 409 basic blocks. Among them, 239 basic blocks contain change points, and 1 520 basic blocks are impacted. In the libxml2 project, there are 75 719 basic blocks. Among them, 130 basic blocks contain change points, and 36 546 basic blocks are impacted. In the lrzip project, there are 6 269 basic blocks. Among them, 245 basic blocks contain change points, and 2 136 basic blocks are impacted. In the mjs project, there are 3 173 basic blocks. Among them, 128 basic blocks contain change points, and 1 818 basic blocks are impacted.

The reason why the basic blocks that contain change points are set as the target is that there are fewer of those blocks than the basic blocks that are impacted. If all impacted basic blocks are set as targets, then many basic blocks will become the targets of fuzzing. For example, in the jasper project, there are a total of 9 467 basic blocks. Of these, 7 112 basic blocks are impacted by change points. If these 7 112 basic blocks are set as targets, the target-guided fuzz testing will degenerate into coverage-guided fuzz testing. Therefore, we choose the basic blocks that contain change points as the target.

The effectiveness of mutation-based fuzz testing highly depends on random mutations. To evaluate the performance of DeltaFuzz, we set the test time to one hour, repeat the experiment three times, and take the average value as the result. The development and the experimental environment are as follows: the operating system is Ubuntu 16.04.7, the CPU is an Intel® Xeon® CPU E5-2678 v3 @ 2.50 GHz, and the amount of RAM is 8 GB.

3.2 Overview of Test Results

Table 2 shows the experimental results of DeltaFuzz and AFLGo, and Table 3 shows the experimental results of DeltaFuzz$_F$ and DeltaFuzz$_B$. They contain three evaluation metrics: the number of test cases executed, the number of crashes triggered, and the number of paths covered. The results are the averages of the three experiments.

In the giflib, jasper and mjs projects, the average number of test cases executed by DeltaFuzz is less than that of AFLGo. In the other three projects, the average number of test cases executed by DeltaFuzz is slightly

Table 2. Comparison of DeltaFuzz, AFLGo and AFLFast in Executed Test Cases, Triggered Crashes and Covered Paths

Project Name	Tool	Number of Excuted Test Cases	Number of Triggered Crashes	Number of Covered Paths
giflib	DeltaFuzz	3 137 569	9	211
	AFLGo	4 674 988	7	217
	AFLFast	5 917 288	13	265
jasper	DeltaFuzz	4 387 472	19	143
	AFLGo	5 261 385	20	152
	AFLFast	107 705	0	6
libming	DeltaFuzz	5 950 227	106	1 515
	AFLGo	5 853 418	88	1 375
	AFLFast	5 620 815	97	1 577
libxml2	DeltaFuzz	5 034 868	116	2 045
	AFLGo	5 012 059	117	1 988
	AFLFast	4 319 284	97	2 020
lrzip	DeltaFuzz	5 942 571	0	2
	AFLGo	5 928 352	0	2
	AFLFast	12 884	0	2
mjs	DeltaFuzz	6 080 262	0	1 180
	AFLGo	7 534 011	0	1 178
	AFLFast	7 181 757	0	1 333
Total	DeltaFuzz	30 532 969	250	5 096
	AFLGo	34 264 213	232	4 912
	AFLFast	23 159 733	207	5 203

Table 3. Comparison of DeltaFuzz$_F$ and DeltaFuzz$_B$ in Executed Test Cases, Triggered Crashes and Covered Paths

Project Name	Tool	Number of Excuted Test Cases	Number of Triggered Crashes	Number of Covered Paths
giflib	DeltaFuzz$_F$	3 185 741	7	199
	DeltaFuzz$_B$	3 342 462	7	215
jasper	DeltaFuzz$_F$	3 777 966	19	140
	DeltaFuzz$_B$	4 166 842	19	131
libming	DeltaFuzz$_F$	5 764 409	94	1 537
	DeltaFuzz$_B$	5 689 859	92	1 449
libxml2	DeltaFuzz$_F$	4 852 108	109	1 993
	DeltaFuzz$_B$	4 615 106	113	1 999
lrzip	DeltaFuzz$_F$	5 657 311	0	2
	DeltaFuzz$_B$	5 741 300	0	2
mjs	DeltaFuzz$_F$	4 263 204	0	1 116
	DeltaFuzz$_B$	6 935 908	0	1 239
Total	DeltaFuzz$_F$	27 500 739	229	4 987
	DeltaFuzz$_B$	30 491 477	231	5 035

more than that of AFLGo. Overall, the total number of test cases executed by DeltaFuzz is 3 731 244 less than that of AFLGo, which is 0.89 times as much as that of AFLGo. This occurs because when DeltaFuzz calculates the fitness values of the test cases during testing, its complexity is slightly higher than that of AFLGo, thus causing additional overhead and resulting in the number of test cases executed by DeltaFuzz being slightly less than that by AFLGo.

The numbers of crashes detected by DeltaFuzz for the jasper and libxml2 projects are both 1 less than those by AFLGo; and both DeltaFuzz and AFLGo trigger the same number of crashes in the lrzip and mjs projects. In the giflib and libming project, the number of crashes triggered by DeltaFuzz is 2 and 18 more than that by AFLGo respectively. In total, DeltaFuzz triggers 18 more crashes than AFLGo, that is, 1.08 times as much as AFLGo. This occurs because the reachability analysis of DeltaFuzz effectively guides fuzz testing, which allows DeltaFuzz to trigger more crashes.

The number of paths covered by DeltaFuzz in the lrzip project is the same as that by AFLGo. In the giflib and jasper projects, the number of paths covered by DeltaFuzz are 1 and 6 less than those by AFLGo, respectively. In the libming, libxml2 and mjs projects, the number of paths detected by DeltaFuzz are 140, 57 and 2 more than those by AFLGo, respectively. In total, DeltaFuzz covers 184 more paths than AFLGo, that is, 1.04 times as much as AFLGo. This occurs because the reachability analysis of DeltaFuzz effectively guides fuzz testing, which allows DeltaFuzz to cover more paths related to change points.

In the giflib and mjs projects, the number of test cases executed by AFLFast is more than that by DeltaFuzz. In the libming, libxml2, jasper and lrzip projects, the number of test cases executed by DeltaFuzz is 329 412, 715 584, 4 279 767 and 5 929 687 more than that by AFLFast, that is, 1.06, 1.17, 40.74 and 461.24 times as much as that by AFLFast, respectively. This occurs because AFLFast has modified score calculating rules for the seed test cases of AFL. If no high-quality test case is generated, AFLFast would give all seed test cases 0 point after a period of time. This means that testing resources would not be allocated to any of the seed test cases, resulting in no new test cases being generated. Therefore, the number of test cases executed in the jasper and lrzip projects for AFLFast is much lower than that for DeltaFuzz and AFLGo.

In the giflib project, the number of crashes triggered by AFLFast is more than that by DeltaFuzz. In the lrzip and mjs projects, neither AFLFast nor DeltaFuzz triggers crashes. In the jasper, libming and libxml2 projects, the number of crashes triggered by AFLFast is less than that by DeltaFuzz. Overall, DeltaFuzz triggers 43 more crashes than AFLFast, up to 1.21 times as much as AFLFast. This occurs because the reachability analysis of DeltaFuzz effectively guides fuzz testing, which allows DeltaFuzz to trigger more crashes.

In the jasper project, the number of paths covered by AFLFast is less than that by DeltaFuzz. In the lrzip project, the number of paths covered by AFLFast is the same as that by DeltaFuzz. In the other four projects,

the number of paths covered by AFLFast is more than that by DeltaFuzz. This occurs because AFLFast concentrates more resources on covering rare branches, and AFLFast covers rarer branches but DeltaFuzz does not; thus, there are more paths covered by AFLFast than DeltaFuzz. However, test cases which cover rare branches may not be able to trigger crashes.

To evaluate the effectiveness of forward, backward, and two-way impact analysis in DeltaFuzz, two strategies, DeltaFuzz$_F$ and DeltaFuzz$_B$, are implemented. In these two strategies, only forward analysis is executed to perform path-sensitive analysis for suspicious basic blocks in DeltaFuzz$_F$, and only backward analysis is executed to perform path-sensitive analysis for suspicious basic blocks in DeltaFuzz$_B$.

For DeltaFuzz$_F$, the total number of test cases executed is less than that of DeltaFuzz, the total number of crashes detected is 21 less than that of DeltaFuzz, and the total number of paths covered is 109 less than that of DeltaFuzz. For DeltaFuzz$_B$, the total number of test cases executed is similar to that of DeltaFuzz, the total number of crashes detected is 19 less than that of DeltaFuzz, and the total number of paths covered is 61 less than that of DeltaFuzz. This occurs because only half of the reachability analysis is executed in DeltaFuzz$_F$ and DeltaFuzz$_B$. During fuzz testing, the target reachability information obtained by DeltaFuzz$_F$ or DeltaFuzz$_B$ is not so complete as the information obtained by DeltaFuzz. Thereby more testing effort is spent on paths that are unrelated to change points.

*Answer for RQ*1. In comparison with AFLGo, DeltaFuzz generates less test cases, but triggers more crashes and covers more paths, 1.08 and 1.04 times as much as AFLGo, respectively. In comparison with AFLFast, DeltaFuzz covers less paths, but triggers more crashes and generates more test cases, 1.21 and 1.32 times as much as AFLFast, respectively.

3.3 Change Point Coverage

In this subsection, we evaluate the change point coverage of DeltaFuzz and AFLGo. Table 4 shows the number of basic blocks that contain change points (BBCCPs) and the number and percentage of BBCCPs that DeltaFuzz, AFLGo and AFLFast cover. Table 4

Table 4. Comparison of DeltaFuzz, AFLGo and AFLFast in Change Point Coverage

Project	Number of BBCCPs	Tool	Covered	Percentage (%)
giflib	46	DeltaFuzz	9	15.25
		AFLGo	9	15.25
		AFLFast	9	15.25
jasper	808	DeltaFuzz	24	2.97
		AFLGo	25	3.09
		AFLFast	14	1.73
lbiming	239	DeltaFuzz	114	47.70
		AFLGo	108	45.19
		AFLFast	108	45.19
libxml2	130	DeltaFuzz	26	20.00
		AFLGo	24	18.46
		AFLFast	24	18.46
lrzip	245	DeltaFuzz	9	3.67
		AFLGo	9	3.67
		AFLFast	9	3.67
mjs	128	DeltaFuzz	50	39.06
		AFLGo	50	39.06
		AFLFast	49	38.28
Total	1 596	DeltaFuzz	232	14.42
		AFLGo	225	13.98
		AFLFast	213	13.24

shows that in the giflib, lrzip and mjs projects, the number of BBCCPs covered by DeltaFuzz is equal to that by AFLGo. In the jasper project, the number of BBCCPs covered by DeltaFuzz is 1 less than that by AFLGo. In the libming and libxml2 projects, the number of BBCCPs covered by DeltaFuzz is 6 and 2 more than that by AFLGo, respectively. In total, DeltaFuzz covers 232 BBCCPs, 1.03 times as much as AFLGo; and the number of BBCCPs covered by DeltaFuzz is 14.42% of the total, 0.44% higher than that by AFLGo. Compared with AFLFast, in the giflib and lrzip projects, the number of BBCCPs covered by DeltaFuzz is the same as that by AFLFast. In the other four projects, the number of BBCCPs covered by DeltaFuzz is greater than that by AFLFast. In total, the number of BBCCPs covered by DeltaFuzz is 19 more than that by AFLFast, 1.09 times as much as that by AFLFast.

To further evaluate the effectiveness of DeltaFuzz in guiding fuzz testing to focus on the change points, two basic blocks that contain change points are randomly selected from each project as the target. Then, the time required by DeltaFuzz, AFLGo, AFLFast and AFL to reach the target basic block is recorded. The timeout is set to one hour, and the experiment is repeated three times to determine the average time. The experimental results are shown in Table 5. T/O means that the target has not been covered after one hour.

As Table 5 shows, when setting the basic block stream.c:1649 or lrzip.c:880 in the lrzip project as the target, none of the three tools can cover the target in one hour. There are two basic blocks that AFLGo can cover faster than DeltaFuzz, AFLFast and AFL, including parser.c:7350 and parser.c:8287 in the libxml2 project. There is one basic block that AFL can cover faster than DeltaFuzz, AFLGo and AFLFast, which is mif_cod.c:493 in the jasper project. There is one basic block that AFLFast can cover faster than DeltaFuzz, AFLGo and AFL, which is gifsponge.c:75 in the giflib project. There are six basic blocks that DeltaFuzz can cover faster than AFLGo, AFLFast and AFL; and they are gifsponge.c:77 in the giflib project, mif_cod.c:487 in the jasper project, parser.c:267 and outputscript.c:551 in the libming project, and mjs.c:7883 and mjs.c:9322 in the mjs project. As Table 5 shows, the time taken by DeltaFuzz to cover the basic block outputscript.c:551 is reduced to 55.16%, 23.66% and 47.90% of that by AFLGo, AFLFast and AFL, respectively. Besides, AFLFast covers none of the target basic blocks in the jasper project.

In general, the time taken for DeltaFuzz, AFLGo, AFLFast and AFL to cover these targets is 19'40", 24'46", 21'48" and 29'11", respectively (T/O excluded). DeltaFuzz is 5'6" and 9'31" faster than AFLGo and AFL, respectively, which is reduced to 20.59% and 32.61% of AFLGo and AFL, respectively. Compared with AFLFast, DeltaFuzz reduces 6'33" to cover target basic blocks (the jasper project is not included because AFLFast runs out of time to cover target basic blocks), and DeltaFuzz takes 30.05% less time than AFLFast. This occurs because DeltaFuzz can reasonably allocate test resources to test cases according to the reachability analysis and fitness function so that the test resources can concentrate more on change points. Thus, DeltaFuzz can cover the target basic block faster than

Table 5. Comparison of the Time Required to Cover Specific Target Basic Blocks

Project	BB's Name	DeltaFuzz	AFLGo	AFLFast	AFL
giflib	gifsponge.c:75	1′ 38″	1′ 25″	0′ 59″	3′ 31″
	gifsponge.c:77	0′ 18″	0′ 39″	0′ 33″	0′ 22″
jasper	mif_cod.c:487	1′ 16″	1′ 51″	T/O	2′ 31″
	mif_cod.c:493	3′ 9″	3′ 16″	T/O	2′ 3″
libming	parser.c:267	2′ 38″	2′ 48″	3′ 25″	2′ 50″
	outputscript.c:551	2′ 51″	5′ 10″	3′ 44″	5′ 57″
libxml2	parser.c:7350	1′ 30″	1′ 13″	2′ 4″	1′ 26″
	parser.c:8287	1′ 31″	1′ 23″	1′ 59″	1′ 24″
lrzip	stream.c:1649	T/O	T/O	T/O	T/O
	lrzip.c:880	T/O	T/O	T/O	T/O
mjs	mjs.c:7883	0′ 53″	0′ 56″	0′ 56″	1′ 4″
	mjs.c:9322	3′ 56″	6′ 5″	8′ 8″	8′ 3″
Total	—	19′ 40″	24′ 46″	21′ 48″	29′ 11″

AFLGo, AFLFast and AFL.

DeltaFuzz can allocate test resources to test cases according to the reachability analysis and fitness function. Thereby the test resources can concentrate more on the change points. As a result, DeltaFuzz can cover more basic blocks that contain change points than AFLGo and cover target basic blocks faster than AFLGo and AFL.

To validate whether DeltaFuzz can concentrate testing resources on BBCCPs, the number of covered BBCCPs during fuzz testing is compared in Fig. 7. Fig. 7 shows the number of BBCCPs covered by DeltaFuzz, AFLGo and AFLFast with different fuzz testing time for the six projects. The horizontal axis is the time of the fuzzing (in seconds), and the vertical axis is the number of BBCCPs covered by the fuzzing tool. Figs.7(a)–7(f) show the number of covered BBCCPs during fuzzing for the six projects, respectively. The table shows that in the lrzip project, DeltaFuzz, AFLGo and AFLFast have similar performance. In the giflib project, DeltaFuzz and AFLFast cover the same number of BBCCPs faster than AFLGo with the same

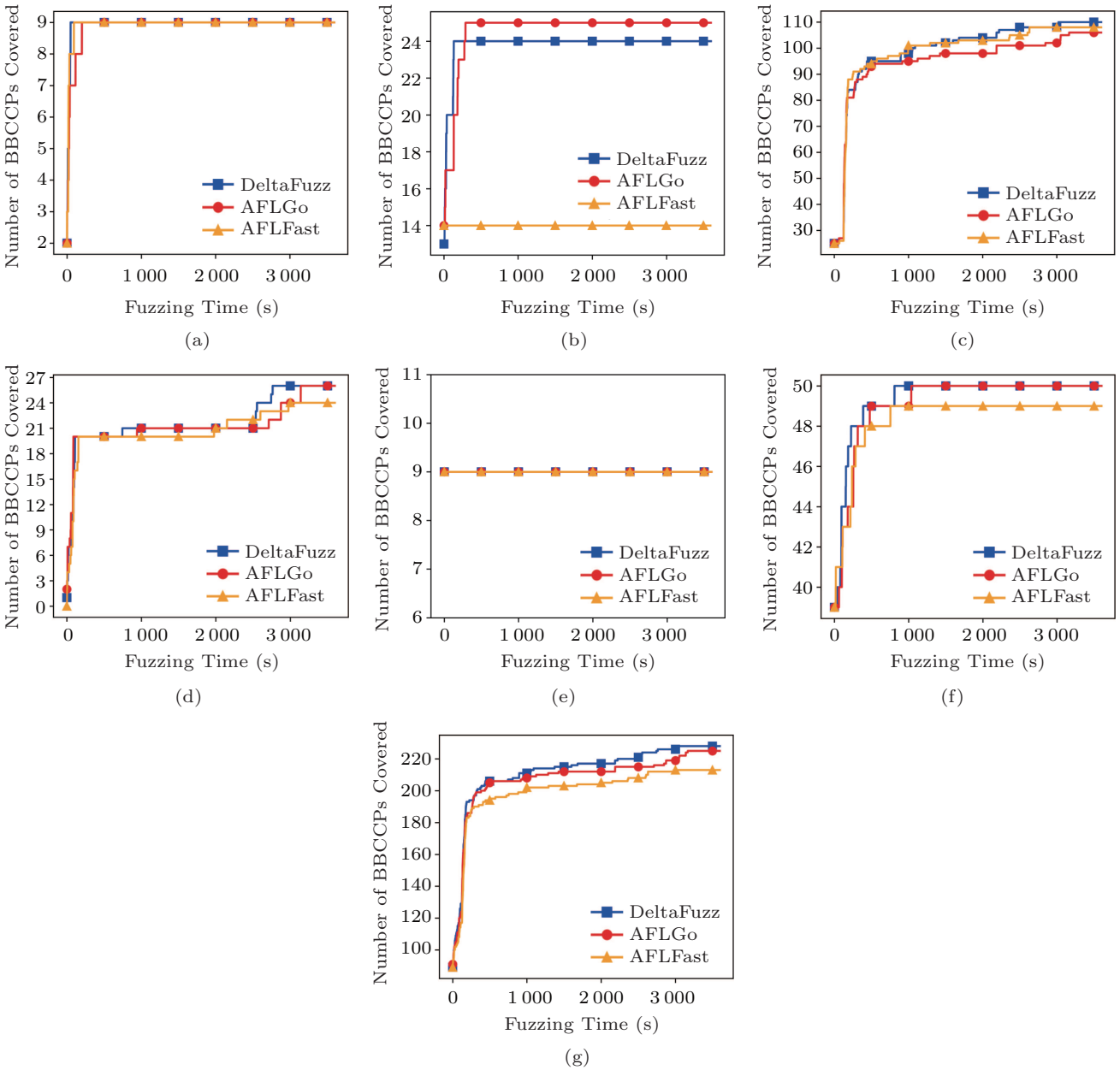


Fig.7. Number of BBCCPs covered by DeltaFuzz, AFLGo and AFLFast with different time. (a) The giflib project. (b) The jasper project. (c) The libming project. (d) The libxml2 project. (e) The lrzip project. (f) The mjs project. (g) All projects.

fuzzing time. In the jasper project, AFLGo covers more BBCCPs than DeltaFuzz and AFLFast with the same fuzzing time. In the libming project, DeltaFuzz covers more BBCCPs than AFLGo and AFLFast with the same fuzzing time. In the libxml2 and mjs projects, DeltaFuzz and AFLGo cover more BBCCPs than AFLFast, and DeltaFuzz covers the same number of BBCCPs faster than AFLGo.

Fig. 7(g) shows the total number of BBCCPs covered during fuzz testing for six projects. It can be seen in Fig.7(g) that when fuzzing for 0–200 seconds, DeltaFuzz, AFLGo and AFLFast cover almost the same number of BBCCPs. After 200 seconds, DeltaFuzz covers more BBCCPs faster than AFLGo and AFLFast at the same fuzzing time.

*Answer for RQ*2. DeltaFuzz outperforms AFLGo and AFLFast in terms of covered BBCCPs. The number of BBCCPs covered by DeltaFuzz is 1.03 and 1.09 times as much as that of AFLGo and AFLFast, respectively. In comparison with the time cost to cover the target basic block, DeltaFuzz reaches the target faster than AFLGo, AFLFast and AFL. Therefore, when a specific change point is set as the target, DeltaFuzz can concentrate more testing resources on the change point.

3.4 Vulnerability Detection

In this subsection, we compare the vulnerabilities detected by DeltaFuzz, AFLGo and AFLFast during the testing process and compare the differences between them. Table 6 shows the number of all vulnerabilities detected by DeltaFuzz, AFLGo and AFLFast during the testing process. In Table 6, subtotal is the number of vulnerabilities detected by different fuzzing tools in each project, while only detected by DeltaFuzz, AFLGo and AFLFast is the number of vulnerabilities that can only be detected by the fuzzing tool, respectively. It also shows that neither DeltaFuzz nor AFLGo detects any vulnerability in the lrzip and mjs projects. Both DeltaFuzz and AFLGo detect 1, 1 and 5 vulnerabilities in the giflib, jasper and libming projects, respectively. In the libxml2 projects, the number of vulnerabilities detected by DeltaFuzz is greater than that by AFLGo, and the vulnerabilities detected by AFLGo can also be detected by DeltaFuzz. In total, DeltaFuzz detects one more vulnerability than AFLGo. Both DeltaFuzz and AFLFast detect 11 vulnerabilities. In the libming and libxml2 projects, DeltaFuzz and AFLFast detect the same number of vulnerabilities. In the giflib project, DeltaFuzz detects one less vulnerability than AFLFast. In the jasper project, AFLFast does not detect any vulnerability while DeltaFuzz detects one vulnerability. Although DeltaFuzz and AFLFast detect the same number of vulnerabilities, DeltaFuzz detects vulnerabilities in four out of the six projects while AFLFast detects vulnerabilities in three out of the six projects. The Venn graph of the vulnerabilities detected is shown in Fig.8.

DeltaFuzz can detect more vulnerabilities than AFLGo. The fitness function and reachability analysis can effectively guide DeltaFuzz to trigger more crashes and reveal more vulnerabilities. Therefore, DeltaFuzz detects more vulnerabilities than AFLGo. Besides, DeltaFuzz detects vulnerabilities in more projects than AFLFast.

*Answer for RQ*3. DeltaFuzz detects 11 vulnerabilities, which is 1 more than AFLGo. In addition, DeltaFuzz can detect all the vulnerabilities that AFLGo detects. Both DeltaFuzz and AFLFast detect 11 vulnerabilities. While DeltaFuzz detects vulnerabilities in four out of the six projects, AFLFast detects vulnerabilities in three out of the six projects. Therefore, DeltaFuzz is more effective than AFLGo and AFLFast in terms of vulnerability detection.

Table 6. Number of Vulnerabilities Detected by DeltaFuzz, AFLGo and AFLFast

Project	DeltaFuzz		AFLGo		AFLFast		All Tools
	Subtotal	Only Detected by DeltaFuzz	Subtotal	Only Detected by AFLGo	Subtotal	Only Detected by AFLFast	
giflib	1	0	1	0	2	1	1
jasper	1	0	1	0	0	0	0
libming	5	0	5	0	5	0	5
libxml2	4	0	3	0	4	0	3
lrzip	0	0	0	0	0	0	0
mjs	0	0	0	0	0	0	0
Total	11	0	10	0	11	1	9

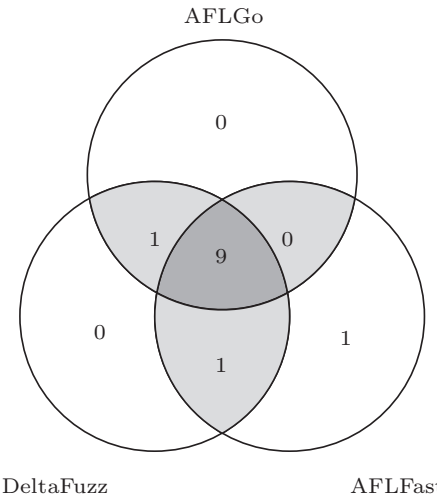


Fig. 8. Venn graph of vulnerability detected by DeltaFuzz, AFLGo and AFLFast.

3.5 Threats to Validity

Internal Validity. First, a common threat to the internal validity of fuzzing methods is the initial seed test cases. In our experiment, both DeltaFuzz and AFLGo are tested by using the same initial seed test case. Regarding the generation of the control flow graph, the graph generation tool used by DeltaFuzz is the same as AFLGo. The subsequent basic block suspicious analysis is also completed based on the basic block call information generated by AFLGo, and the entire process is automated. In addition, because the mutation of the fuzz testing is random, randomly generating test cases may have some impact on the effectiveness of the test. In this paper, the experiments are repeated three times, and every project is tested for one hour in each run of the experiment. Then, the average values are taken to reduce the impact of randomness. To validate whether DeltaFuzz can concentrate testing resources on change points, two basic blocks that contain change points are selected randomly as targets. However, the random selection of basic blocks may introduce uncertainties. Therefore, another experiment is conducted on the coverage of basic blocks that contain change points to reduce the impact of random selection.

External Validity. The giflib, jasper, libming, libxml2, lrzip and mjs projects are removed from the 11 experimental objects used by AFLGo because the historical versions of the six projects are available for us to analyze the impact of changes. These projects are well-known open-source projects and are widely adopted in the community [20, 29–33]. Based on the versatility and diversity of these projects, we believe that the research results and conclusions in the paper can be transferred to similar tools.

Construction Validity. In the experiment, five aspects of DeltaFuzz are evaluated: test cases executed, crashes triggered, paths covered, vulnerabilities detected, and change points covered. Among these aspects, the first four metrics are commonly used in fuzz testing. In addition, we use the change point coverage to validate the effectiveness of DeltaFuzz. This metric is especially important for regression testing [33, 34] because it can be used to check whether fuzz testing concentrates testing resources on the impacted code.

4 Related Work

According to the analytical depth of the internal logic of the source code, fuzz testing can be divided into black-box fuzz testing, white-box fuzz testing and grey-box fuzz testing. Historical version information guided fuzz testing belongs to grey-box fuzz testing. Grey-box fuzz testing can be divided into coverage-guided fuzz testing and target-guided fuzz testing according to the guidance method. In this section, we introduce related work for these two categories.

4.1 Coverage-Guided Fuzz Testing

The purpose of coverage-guided fuzz testing is to generate test cases that can cover new code segments as many as possible during testing to achieve high code coverage.

In 2013, Zalewski released the fuzz testing tool AFL. AFL is a commonly used coverage-guided fuzz testing tool. It uses slight instrumentation and genetic algorithms to effectively improve code coverage and the quality of test cases. To date, AFL has exposed many important security vulnerabilities in dozens of software projects, including PHP and Firefox. AFL has gained popularity among testing researchers due to its excellent versatility, scalability and rapid deployment. Currently, many fuzz testing tools have been developed on the foundation of AFL.

In 2019, Nagy and Hicks [35] proposed UnTracer, a tracker for coverage guidance. To save the cost of tracking test cases during fuzz testing, UnTracer improves the instrumentation algorithm. UnTracer is evaluated on eight real software programs that are commonly used in the fuzzing community. The experiments show that compared with other similar tools such as AFL-Clang and AFL-QEMU, UnTracer has the lowest cost in tracing test cases.

In 2019, Liang *et al.* developed DeepFuzzer [36], which is an enhanced grey-box fuzz testing tool that generates high-quality initial seed test cases through symbolic execution. DeepFuzzer adopts a hybrid mutation strategy when mutating seed test cases, which can balance the mutation frequency between different seeds to improve the efficiency of fuzz testing. The experimental results show that DeepFuzzer can cover more paths and find more vulnerabilities during the same time period than AFL, AFLFast, QSYM and other methods.

In summary, coverage-guided fuzz testing approaches usually focus on making some improvements to cover more paths at the same time or to reduce the time costs of achieving the same coverage [37–40]. For example, fuzz testing can be combined with symbolic execution to obtain high-quality initial seed test cases generated by symbolic execution to improve the path coverage of fuzz testing [41] or to improve the instrumentation algorithm of fuzz testing to reduce instrumentation overhead during the testing process [42]. When there is no clear target, coverage-guided fuzz testing is effective because it can fully test an entire program. When using static analysis, defect prediction or other methods to acquire the location of possible vulnerabilities in the program, we usually hope that the testing resource is concentrated on the location of possible vulnerabilities. Therefore, some research has sought to introduce targets as guidance to further improve the efficiency of fuzz testing [43].

4.2 Target-Guided Fuzz Testing

The purpose of target-guided fuzz testing is to generate test cases that can cover the target code block or target function as many as possible during the testing so that resources can be concentrated on testing the target code.

In 2017, Böhme *et al.* [20] developed a directed grey-box fuzzing tool AFLGo. AFLGo calculates the distance between each basic block and the target basic block in advance before starting fuzz testing, generates the corresponding instrumentation file to insert the distance during instrumentation, and integrates the simulated annealing energy plan to generate new test cases. AFLGo allocates more energy to test cases that are close to the target basic block, and reduces the energy of test cases that are far away from the target basic block. The experimental results show that AFLGo can cover the target basic blocks faster than directed white-box fuzzing tools and nondirected grey-box fuzzing.

In 2018, Lemieux and Sen [44] implemented FairFuzz based on AFL to cover the rare branches of the program being tested as many as possible. FairFuzz automatically identifies the rare branches covered by the inputs of AFL and uses a novel mutation mask creation algorithm to make the test cases generated by the mutation more likely to cover the rare branches. The mutation mask is dynamically calculated during fuzz testing, thereby it can be adapted to different test targets. The experimental results show that FairFuzz has a higher branch coverage rate than AFL and covers more rare branches during the same time period.

In 2018, Chen *et al.* [24] implemented Hawkeye, which is a target-guided grey-box fuzzing tool. Before fuzz testing starts, Hawkeye uses static analysis on the program being tested and calculates the distance based on the call graph and the control flow graph. During fuzz testing, Hawkeye generates dynamic metrics based on both static information and seed execution traces and uses it for seed prioritization, power scheduling, and adaptive mutating to help Hawkeye reach the target faster. Fifteen CVE (Common Vulnerabilities and Exposures) were detected by Hawkeye in the experiments.

In 2019, Situ *et al.* [45] implemented TAFL to improve the variation and energy distribution of AFL. Based on the static semantic metrics of the program being tested, TAFL instructs the fuzz testing tool to move toward areas with higher probabilities of containing vulnerabilities. In addition, granularity-aware scheduling of mutators, which dynamically allocates ratios of different mutation operators, was proposed. The experimental results showed that TAFL discovered new security vulnerabilities and exposed three CVE.

In 2019, Böhme and Pham [46] implemented AFLFast based on the coverage-guided grey-box fuzzing tool AFL. AFLFast uses the Markov chain model to determine the testing resources allocated to the seed test case, which makes fuzz testing more inclined to cover low-frequency paths. Experiments showed that AFLFast exposes CVE much faster than AFL, and it triggers more unique crashes than AFL.

In 2020, Wüstholz and Christakis [8] proposed a novel target-guided grey-box fuzzing technique, which uses online static analysis to guide the fuzz testing tool to cover a set of target positions. The effectiveness of this technique was evaluated on 27 real-world projects by extending the most advanced Ethereum smart contract fuzz testing tool, Harvey [47]. The experimental results showed that the performance of the proposed

target-guided grey-box fuzz testing technique is significantly better than that of Harvey.

In target-guided fuzz testing tools, one or more targets are specified, and the target is usually a basic block or a function [20]. These targets are generally determined by static analysis or other techniques. Unlike coverage-guided fuzz testing, target-guided fuzz testing can achieve better performance during the same time period, even if high-quality initial seed test cases are not provided [32]. Although target-guided fuzzing tests are more effective than coverage-guided fuzzing tests, they usually focus on one or more basic blocks rather than paths [20]. If the path coverage is used as the criterion for calculating fitness, the effectiveness of fuzz testing can be further improved.

On the basis of target-guided fuzz testing, some studies have proposed path-sensitive fuzz testing. In 2018, Gan *et al.* [48] proposed a path-sensitive fuzz testing scheme CollAFL. CollAFL reduces path collisions by providing more accurate coverage information while maintaining low instrumentation overhead. It also uses coverage information to implement three fuzz testing strategies to improve the speed of finding vulnerabilities and paths. The experimental results show that CollAFL can reduce the ratio of edge collisions to nearly zero in projects with common path collision. It can also cover more paths and find more vulnerabilities than AFL during the same time period.

Although CollAFL also conducts path-sensitive fuzz testing, unlike DeltaFuzz, CollAFL still conducts coverage-guided fuzz testing. More accurate coverage information enables CollAFL to distinguish different paths more accurately [48]. To the best of our knowledge, DeltaFuzz is the first path-sensitive fuzz testing tool that calculates fitness through suspicious basic blocks, and it is also the first fuzz testing tool guided by historical version information.

Target-guided fuzz testing tools, such as AFLGo, usually set locations of potential or known vulnerabilities as targets of fuzz testing to guide test case generation. Although AFLGo also achieves good results, it is usually difficult to get priori knowledge about the location of vulnerabilities before testing. By contrast, updates in the process of software evolution are identified as change points and set as targets for DeltaFuzz. Change points can be automatically obtained by difference comparison, which makes DeltaFuzz more scalable and applicable in regression testing. Besides, AFLFast sets rare branches as targets of fuzz testing, but rare branches probably exist in the last version and are unrelated to version updates. As a result, AFLFast is not suitable to be applied in the scenario of regression testing. In addition, DeltaFuzz is a path-sensitive fuzz testing approach, and the fitness of the test case is calculated based on its executing trace to guide test case generation.

5 Conclusions

In this paper, we proposed a path-sensitive fuzz testing method guided by historical version information and implemented a prototype tool DeltaFuzz. DeltaFuzz acquires the change points by comparing the differences between the program being tested and its last version and then performs change impact analysis to obtain a set of suspicious basic blocks. During the testing process, DeltaFuzz analyzes the execution path of the test case, calculates its fitness based on the suspicious basic block set, and allocates resources to the test case according to the fitness value. As the experimental result shows, DeltaFuzz improves the number of crashes triggered and the number of BBCCPs covered, which are 1.07 and 1.03 times as many as those of AFLGo, 1.20 and 1.09 times as many as those of AFLFast, respectively. Regarding covering specific targets, DeltaFuzz covers them faster than AFLGo, AFLFast and AFL, and the time taken by DeltaFuzz is reduced by 20.59%, 30.05% and 32.61%, respectively.

In future work, symbolic execution can be integrated into DeltaFuzz to improve the quality of initial seed test cases so that DeltaFuzz can cover change points in a shorter time and find deeper vulnerabilities. In addition, the fitness function can also be adapted to other algorithms, such as the simulated annealing algorithm or particle swarm optimization algorithm based on Monte Carlo simulation, to allocate test resources for test cases more accurately.

References

[1] Masso J, Pino F J, Pardo C *et al.* Risk management in the software life cycle: A systematic literature review. *Computer Standards & Interfaces*, 2020, 71: Article No. 103431. DOI: 10.1016/j.csi.2020.103431.

[2] Gu T, Ma X, Xu C *et al.* Synthesizing object transformation for dynamic software updating. In *Proc. the 39th IEEE/ACM International Conference on Software Engineering Companion*, May 2017, pp.336-338. DOI: 10.1109/ICSE-C.2017.96.

[3] Khatibsyarbini M, Isa M A, Jawawi D N A *et al.* Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 2018, 93: 74-93. DOI: 10.1016/j.infsof.2017.08.014.

[4] Han J C, Zhou Z Q. Metamorphic fuzz testing of autonomous vehicles. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, June 27–July 19, 2020, pp.380-385. DOI: 10.1145/3387940.3392252.

[5] Dong Z, Böhme M, Cojocaru L *et al.* Time-travel testing of Android apps. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, June 27–July 19, 2020, pp.481-492. DOI: 10.1145/3377811.3380402.

[6] Qian J, Zhou D. Prioritizing test cases for memory leaks in Android applications. *Journal of Computer Science and Technology*, 2016, 31(5): 869-882. DOI: 10.1007/s11390-016-1670-2.

[7] Chen Y, Su T, Su Z. Deep differential testing of JVM implementations. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.1257-1268. DOI: 10.1109/ICSE.2019.00127.

[8] Wüstholz V, Christakis M. Targeted greybox fuzzing with static lookahead analysis. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, June 27–July 19, 2020, pp.789-800. DOI: 10.1145/3377811.3380388.

[9] Böhme M, Manès V J M, Cha S K. Boosting fuzzer efficiency: An information theoretic perspective. In *Proc. the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020, pp.678-689. DOI: 10.1145/3368089.3409748.

[10] Song S, Song C, Jang Y *et al.* CrFuzz: Fuzzing multi-purpose programs through input validation. In *Proc. the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020, pp.690-700. DOI: 10.1145/3368089.3409769.

[11] Havrikov N. Efficient fuzz testing leveraging input, code, and execution. In *Proc. the 39th IEEE/ACM International Conference on Software Engineering Companion*, May 2017, pp.417-420. DOI: 10.1109/ICSE-C.2017.26.

[12] Klees G, Ruef A, Cooper B *et al.* Evaluating fuzz testing. In *Proc. the 2018 ACM SIGSAC Conference on Computer and Communications Security*, October 2018, pp.2123-2138. DOI: 10.1145/3243734.3243804.

[13] Zhang M Z, Gong Y Z, Wang Y W *et al.* Unit test data generation for C using rule-directed symbolic execution. *Journal of Computer Science and Technology*, 2019, 34(3): 670-689. DOI: 10.1007/s11390-019-1935-7.

[14] He J, Balunović M, Ambroladze N *et al.* Learning to fuzz from symbolic execution with application to smart contracts. In *Proc. the 2019 ACM SIGSAC Conference on Computer and Communications Security*, November 2019, pp.531-548. DOI: 10.1145/3319535.3363230.

[15] Zhang Q, Wang J, Gulzar M A *et al.* BigFuzz: Efficient fuzz testing for data analytics using framework abstraction. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering*, September 2020, pp.722-733. DOI: 10.1145/3324884.3416641.

[16] Nguyen H L, Nassar N, Kehrer T *et al.* MoFuzz: A fuzzer suite for testing model-driven software engineering tools. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering*, September 2020, pp.1103-1115. DOI: 10.1145/3324884.3416668.

[17] Olsthoorn M, Van Deursen A, Panichella A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering*, September 2020, pp.1224-1228. DOI: 10.1145/3324884.3418930.

[18] Nguyen T D, Pham L H, Sun J *et al.* sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proc. the 42nd ACM/IEEE International Conference on Software Engineering*, June 27–July 19, 2020, pp.778-788. DOI: 10.1145/3377811.3380334.

[19] Manès V J M, Kim S, Cha S K. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proc. the 42nd ACM/IEEE International Conference on Software Engineering*, June 27–July 19, 2020, pp.1024-1036. DOI: 10.1145/3377811.3380421.

[20] Böhme M, Pham V T, Nguyen M D *et al.* Directed greybox fuzzing. In *Proc. the 2017 ACM SIGSAC Conference on Computer and Communications Security*, October 30–November 3, 2017, pp.2329-2344. DOI: 10.1145/3133956.3134020.

[21] Gao X, Saha R K, Prasad M R *et al.* Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, June 27–July 19, 2020, pp.1147-1158. DOI: 10.1145/3377811.3380415.

[22] Wen C, Wang H, Li Y *et al.* MemLock: Memory usage guided fuzzing. In *Proc. the 42nd ACM/IEEE International Conference on Software Engineering*, June 27–July 19, 2020, pp.765-777. DOI: 10.1145/3377811.3380396.

[23] Babić D, Bucur S, Chen Y *et al.* FUDGE: Fuzz driver generation at scale. In *Proc. the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, August 2019, pp.975-985. DOI: 10.1145/3338906.3340456.

[24] Chen H, Xue Y, Li Y *et al.* Hawkeye: Towards a desired directed grey-box fuzzer. In *Proc. the 2018 ACM SIGSAC Conference on Computer and Communications Security*, October 2018, pp.2095-2108. DOI: 10.1145/3243734.3243849.

[25] Medicherla R K, Komondoor R, Roychoudhury A. Fitness guided vulnerability detection with greybox fuzzing. In *Proc. the 42nd IEEE/ACM International Conference on Software Engineering*, June 27–July 19, 2020, pp.513-520. DOI: 10.1145/3387940.3391457.

[26] Österlund S, Razavi K, Bos H *et al.* ParmeSan: Sanitizer-guided greybox fuzzing. In *Proc. the 29th USENIX Security Symposium*, August 2020, pp.2289-2306.

[27] Gao X, Mechtaev S, Roychoudhury A. Crash-avoiding program repair. In *Proc. the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2019, pp.8-18. DOI: 10.1145/3293882.3330558.

[28] Fioraldi A, Maier D, Eißfeldt H *et al.* AFL++: Combining incremental steps of fuzzing research. In *Proc. the 14th USENIX Workshop on Offensive Technologies*, August 2020.

[29] Grieco G, Ceresa M, Buiras P. QuickFuzz: An automatic random fuzzer for common file formats. *ACM SIGPLAN Notices*, 2016, 51(12): 13-20. DOI: 10.1145/2976002.2976017.

[30] Liang H, Zhang Y, Yu Y *et al.* Sequence coverage directed greybox fuzzing. In *Proc. the 27th IEEE/ACM International Conference on Program Comprehension*, May 2019, pp.249-259. DOI: 10.1109/ICPC.2019.00044.

[31] Zhang M, Liu J, Ma F *et al.* IntelliGen: Automatic driver synthesis for fuzz testing. In *Proc. the 43rd IEEE/ACM International Conference on Software Engineering*, May 2021, pp.318-327. DOI: 10.1109/ICSE-SEIP52600.2021.00041.

[32] You W, Liu X, Ma S *et al.* SLF: Fuzzing without valid seed inputs. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.712-723. DOI: 10.1109/ICSE.2019.00080.

[33] Choi W, Sen K, Necul G *et al.* DetReduce: Minimizing Android GUI test suites for regression testing. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering*, May 27–June 3, 2018, pp.445-455. DOI: 10.1145/3180155.3180173.

[34] Zhang L. Hybrid regression test selection. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering*, May 27–June 3, 2018, pp.199-209. DOI: 10.1145/3180155.3180198.

[35] Nagy S, Hicks M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proc. the 2019 IEEE Symposium on Security and Privacy*, May 2019, pp.787-802. DOI: 10.1109/SP.2019.00069.

[36] Liang J, Jiang Y, Wang M *et al.* DeepFuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2019, 18(6): 2675-2688. DOI: 10.1109/TDSC.2019.2961339.

[37] Fioraldi A, D'Elia D C, Coppa E. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proc. the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2020, pp.1-13. DOI: 10.1145/3395363.3397372.

[38] Chen Y, Poskitt C M, Sun J *et al.* Learning-guided network fuzzing for testing cyber-physical system defences. In *Proc. the 34th IEEE/ACM International Conference on Automated Software Engineering*, November 2019, pp.962-973. DOI: 10.1109/ASE.2019.00093.

[39] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May 2018, pp.697-710. DOI: 10.1109/SP.2018.00056.

[40] Padhye R, Lemieux C, Sen K. JQF: Coverage-guided property-based testing in Java. In *Proc. the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2019, pp.398-401. DOI: 10.1145/3293882.3339002.

[41] Noller Y, Kersten R, Păsăreanu C S. Badger: Complexity analysis with fuzzing and symbolic execution. In *Proc. the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2018, pp.322-332. DOI: 10.1145/3213846.3213868.

[42] Zhou C, Wang M, Liang J *et al.* Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering*, September 2020, pp.858-870. DOI: 10.1145/3324884.3416572.

[43] Wang T, Wei T, Gu G *et al.* TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proc. the 2010 IEEE Symposium on Security and Privacy*, May 2010, pp.497-512. DOI: 10.1109/SP.2010.37.

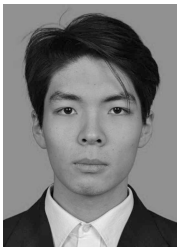
[44] Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, September 2018, pp.475-485. DOI: 10.1145/3238147.3238176.

[45] Situ L, Wang L, Li X, Guan L, Zhang W, Liu P. Energy distribution matters in greybox fuzzing. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.270-271. DOI: 10.1109/ICSE-Companion.2019.00109.

[46] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. *IEEE Transactions on Software Engineering*, 2017, 45(5): 489-506. DOI: 10.1109/TSE.2017.2785841.

[47] Wüstholz V, Christakis M. Harvey: A greybox fuzzer for smart contracts. In *Proc. the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020, pp.1398-1409. DOI: 10.1145/3368089.3417064.

[48] Gan S, Zhang C, Qin X *et al.* CollAFL: Path sensitive fuzzing. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May 2018, pp.679-696. DOI: 10.1109/SP.2018.00040.



Jia-Ming Zhang received his Bachelor's degree in software engineering from Beijing Information Science and Technology University, Beijing, in 2020. He is currently studying for his Master's degree at Beijing Information Science and Technology University, Beijing. His research interests include software analysis and testing.

Zhan-Qi Cui received his B.E. degree in software engineering and his Ph.D. degree in computer software and theory, in 2005 and 2011, respectively, both from Nanjing University, Nanjing. He was a visiting Ph.D. student in University of Virginia, Virginia, from Sept. 2009 to Sept. 2010. He is currently an associate professor at Beijing Information Science and Technology University, Beijing. His research interests include software analysis and testing.

Xiang Chen received his B.S. degree in management from Xi'an Jiaotong University, Xi'an, in 2002, and his M.Sc. and Ph.D. degrees in computer software and theory from Nanjing University, Nanjing, in 2008 and 2011, respectively. He is currently an associate professor with the School of Information Science and Technology, Nantong University, Nantong. His research interests mainly include software maintenance and software testing, such as security vulnerability prediction, software defect prediction, combinatorial testing, regression testing, and fault localization.

Huan-Huan Wu received her Bachelor's degree in software engineering from Bengbu University, Bengbu, in 2020. She is currently studying for her Master's degree at Beijing Information Science and Technology University, Beijing. Her research interests include software analysis and testing.

Li-Wei Zheng received his Ph.D. degree in computer software and theory from Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, in 2009. He is currently an associate professor at Beijing Information Science and Technology University, Beijing. His research interests include requirement engineering and trusted computing.

Jian-Bin Liu received his Ph.D. degree in computer software and theory from Northwestern University, Xi'an, in 2004. He is currently a professor at Beijing Information Science and Technology University, Beijing. His research interests include program modeling theory and methods, intelligent software development technology, and model-driven software engineering.