applied sciences

MDPI

Article

SwitchFuzz: Switch Short-Term Goals in Directed Grey-Box Fuzzing

Ziheng He, Peng Jia *, Yong Fang, Yuying Liu and Hairu Luo

School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China
* Correspondence: pengjia@scu.edu.cn

Abstract: In recent years, fuzzing has become a powerful tool for security researchers to uncover security vulnerabilities. It is used to discover software vulnerabilities by continuously generating malformed inputs to trigger bugs. Directed grey-box fuzzing has also been widely used in the verification of patch testing and in vulnerability reproduction. For directed grey-box fuzzing, the core problem is to make test cases reach the target and trigger vulnerabilities faster. Selecting seeds that are closer to the target site to be mutated first is an effective method. For this purpose, the DGF calculates the distance between the execution path and the target site by a specific algorithm. However, as time elapses in the execution process, the seeds covering a larger amount of basic blocks may be overlooked due to their long distances. At the same time, directed fuzzing often ignores the impact of coverage on test efficiency, resulting in a local optimum problem without accumulating enough valuable test cases. In this paper, we analyze and discuss these problems and propose SwitchFuzz, a fuzzer that can switch short-term goals during execution. SwitchFuzz keeps shortening the distance of test cases to reach the target point when it performs well and prioritizes reaching the target point. When positive feedback is not achieved over a period of time, SwitchFuzz tries to explore more possibilities. We compared the efficiency of SwitchFuzz with that of AFLGO in setting single target and multiple targets for crash recurrence in our experiments, respectively. The results show that SwitchFuzz produces a significant improvement over AFLGO in both the speed and the probability of triggering a specified crash. SwitchFuzz can discover more edges than AFLGO in the same amount of time and can generate seeds with smaller distances.

Keywords: fuzzing; directed fuzzing; vulnerability detection; crash reproduce

check for updates

Citation: He, Z.; Jia, P.; Fang, Y.; Liu, Y.; Luo, H. SwitchFuzz: Switch Short-Term Goals in Directed Grey-Box Fuzzing. *Appl. Sci.* **2022**, *12*, 11097. https://doi.org/10.3390/app122111097

Academic Editors: Marek Pawlickie and Rafał Kozik

Received: 14 October 2022
Accepted: 31 October 2022
Published: 2 November 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In recent years, fuzzing has stood out in vulnerability mining. It sends a large number of malformed inputs to the target program to trigger crashes. Grey-box fuzzing (GF), represented by AFL [1], generates code when compiling to collect information. It involves the use of genetic algorithms, turning the problem of generating test cases into using genetic algorithms for numerical optimization of code coverage. Research into fuzzing has made significant progress. There are now many fuzzing testing tools based on AFL.

GF cannot be effectively targeted. That is, it cannot allow the fuzzing test to reach the part of the target code that the tester needs to test. Directed fuzzing (DF) is designed for this purpose. It spends most of its time reaching the designated target location without spending a lot of resources on unrelated paths. This has great practical significance in areas such as patch testing, crash recurrence, static analysis report verification and information flow detection.

There are currently two forms of DF. One is directed symbolic execution (DSE), which converts the path-reachability problem into a constraint-solving problem to generate test cases that can reach the target point. The other is directed grey-box fuzzing (DGF), which transforms the path-reachability problem into a distance optimization problem and continuously generates seeds closer to the goal using optimization algorithms.

AFL can also be regarded as a guided fuzzing test, but it is based on coverage-guided, the goal of which is to explore as many code fragments as possible to thoroughly test the target program. In contrast to the AFL coverage-based guidance method, DGF, represented by AFLGo [2], uses distance-based guidance. The distance from each basic block and function to the target point is calculated during compilation and the seed distance is calculated dynamically at runtime. Subsequently, it selectively schedules the seeds based on their distance and other indicators.

To evaluate the quality of a directed grey-box fuzzer, the most important indicator is whether it can trigger a crash that reaches the specified target point in a shorter amount of time. This is our purpose for using it. Therefore, the time to reach the target point naturally becomes a key indicator. DGF often pays little attention to coverage. If multiple paths can reach the target point, most DGF methods tend to ignore some paths. These paths may meet the necessary conditions to trigger the vulnerability. (We refer to seeds that cover more basic blocks as "long path seeds" in the following sections). However, spending time exploring more paths inevitably leads to a decrease in time efficiency. How to balance the time and coverage for reaching the target point is a problem that DGF needs to solve.

DGF often requires the user to set a time for accumulating sufficient test cases (described in detail in Section 3.2) though the user is often not able to set this time properly. If the time is set too short, the fuzzer cannot accumulate sufficient seeds of value. This can result in a subsequent blind pursuit of distance reduction for a certain path during testing, thus falling into the path local optimum problem and leading to crashes that are more difficult to trigger. However, too long time setting can affect the efficiency of targeting, which can be fatal for DGF.

In this paper, we analyze and discuss these issues. We implement SwitchFuzz, a fuzzer that switches short-term goals during execution, based on AFLGo. SwitchFuzz gives priority to reaching the target point when the test state is good and keeps shortening the distance of test cases to the target point. When it does not receive better feedback in a particular period, it will try to explore further possibilities. This addresses the shortcomings resulting from unreasonable time settings and avoids the path exploration falling into a local optimum. SwitchFuzz also addresses the issue that basic blocks with vulnerability trigger prerequisites located on long paths that are ignored. In addition, we optimize the seed scheduling policy in different situations.

To evaluate the performance of SwitchFuzz, we compared it with AFLGo on several real program datasets. In experiments undertaken with separate single-target and multi-target setups, SwitchFuzz produced a significant improvement in the probability and speed of triggering a specified crash within a certain time. In addition, SwitchFuzz was able to find more edges and to generate seeds with smaller distances than AFLGo in the same test time.

The main contributions of this paper are summarized as follows:

(1) We summarize the current understanding of directed grey-box fuzzing and present four problems to be solved or improved.

(2) We propose a method to allocate queues according to tasks, which can reach the target point faster and preserve path diversity.

(3) We propose a method to solve the contradiction between arrival speed and multi-path coverage, which can complete the appropriate task at the appropriate stage.

(4) We apply these methods to a new directed grey-box fuzzing tool named SwitchFuzz and comprehensively evaluate the time required for vulnerability triggering and path coverage.

2. Background

In this Section, we provide a short introduction to the background of DGF. In Section 2.1, we introduce calculation of the DGF distance. In Section 2.2, we introduce the seed queue and the scheduling of seeds.

2.1. Distance

The purpose of directed grey-box fuzzing (DGF) is to test a specific target code. The tester sets the target points before executing the test. The fuzzer then continuously optimizes the input test cases. The execution path of the generated test cases gradually approaches the target point and triggers the corresponding vulnerability. Most DGF methods obtain the function call graph (CG) and control flow graph (CFG) of the target program during the static analysis phase. The distance from each basic block to the target point is calculated using CG and CFG and the distance is inserted into the basic block at compile time. During fuzzing, the DGF calculates the distance to the target point based on the execution path of the test case.

An example of AFLGo, which is the classic DGF, is presented below. Figure 1 illustrates the control flow graph and T1 and T2 are the target basic blocks. The A-B-D path is not reachable by the target point, so the distance is infinite. For the basic block A, the closest execution path to T1 is A-B-C-T1, with three edges in the path. The closest execution path to T2 is A-B-T2, with two edges. The distance of the basic block is calculated as the harmonic mean of the distance to each target. Considering the basic block A as an example, its distance is $(\frac{1}{2} + \frac{1}{3})^{-1} = \frac{6}{5}$. The distance of a test case is calculated as the average distance of all basic blocks of its route. If a test case is entered, the program executes A-B-C-T1-T2 in sequence. The distance of this test case is $(\frac{6}{5} + \frac{2}{3} + \frac{2}{3} + 0 + 0)/5 = 0.507$. If the program execution path after another test case input is A-B-T2, the distance is $(\frac{6}{5} + \frac{2}{3} + 0)/3 = 0.622$. In this way, for each input, a distance value is obtained after the execution based on its execution result.

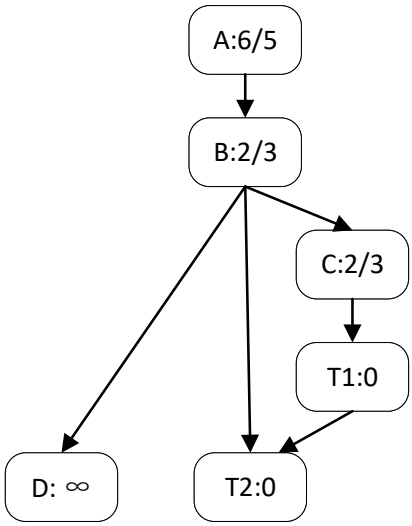

Figure 1. Example control-flow graph with DGF distances specified on each basic block.

2.2. Seed Queue

Like other grey-box fuzzing, the workflow of directed fuzzing involves continuously attempting to mutate the input test cases and collection of program information. When test cases with positive feedback (e.g., distance reduction) are obtained, the fuzzer keeps them as seeds and stores them in the seed queue. When needed, the fuzzer selects out preferred seeds to mutate. For directed fuzzing, when the input is executed and the distance is found to be smaller than the current minimum distance, or new coverage is found, the input is stored as a seed in the seed queue and marked as "favored". This marker is used to give a higher priority to the seed in the next seed scheduling step.

Most of the DGFs based on AFLGo use an annealing algorithm. They divide the entire fuzzing execution period into exploration and exploitation phases. The exploration time is set by the user. Seeds are given a score based on their distance. The higher the score the more energy the seed will gain. Seeds with higher energy are given more chances to mutate. At the beginning of the exploration phase, the seeds in the seed queue are considered to

be of equal value and are given the same energy. The fuzzer accumulates as many test cases as possible and explores more of the program state space. As the exploitation phase is entered, the energy of seeds at larger distances decreases and eventually they are no longer noticeable. It is worth noting that the actual distances are normalized according to the maximum and minimum distances among all seeds.

The curves for energy variation with time for each of SeedA and SeedB with their normalized distances held constant are shown in Figure 2. It is important to note that the energy describes the level of attention given to the seed; we normalized the range of energy to 0 to 1 for ease of presentation. The reason for the existence of an upper limit is that, no matter how valuable the fuzzing system considers this seed to be, it will still go through the full mutation process. For cases where the seed energy is 0, the fuzzing system will hardly perform the mutation operation. The distance for SeedA is smaller, while the distance for SeedB is larger. The energy of SeedB decreases with time and gradually approaches zero after entering the exploitation phase. The energy of SeedA, on the other hand, is the opposite and gradually approaches to one. Using this mechanism, the DGF can continuously optimize the test cases and gradually generate seeds with smaller distances.

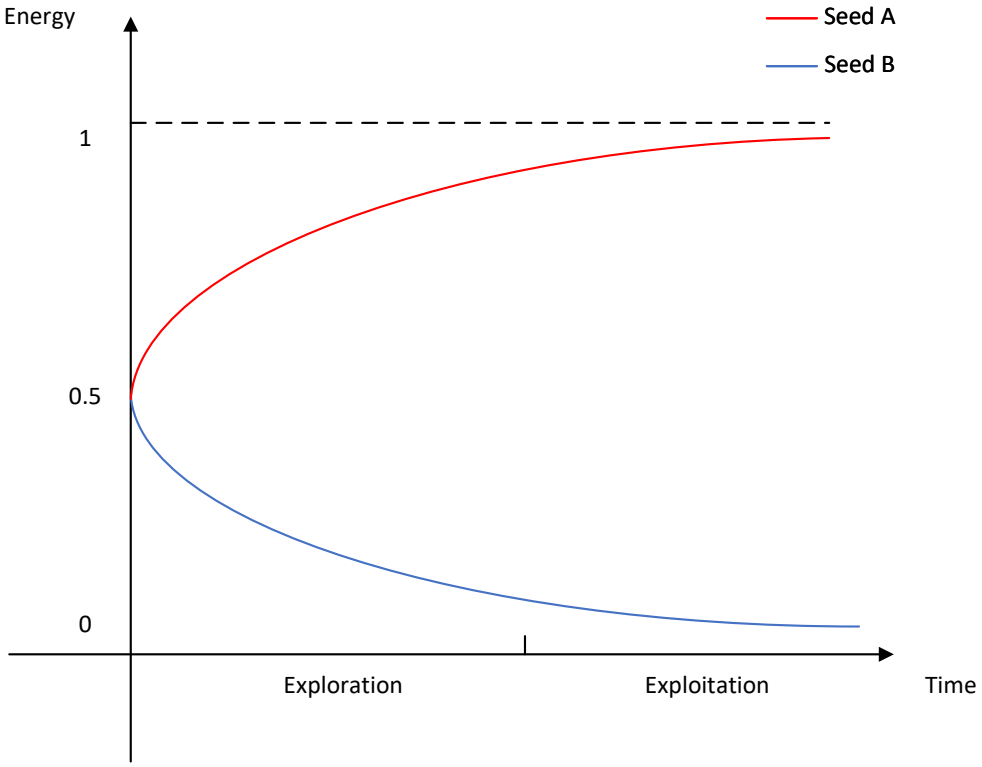

Figure 2. Variation of seed energy with time.

2.3. Skip Mechanism

There are a finite number of seeds in the queue. When no new seeds are produced and execution reaches the end of the queue, the fuzzer will redirect the pointer of the selected seed to the head of the queue and start a new round of mutation. At this point, if a new seed is produced, it is often worth more than the previous seed. Fuzzers such as AFLGo tend to give seeds a number of markers to help determine whether they are valuable seeds or not.

Specifically, the *was_fuzzed* flag indicates whether the seed has been used. The *favored* flag indicates whether the seed should be noticed, which is usually the case for newly generated seeds. The *pending_favored* flag indicates whether there are seeds noticed in the seed queue. AFLGo determines the skip probability based on the *was_fuzzed* flag, the *favored* flag, and the global *pending_favrored* flag of the current seed. If a seed of interest is found globally, *pending_favored* is set to one, at which point seeds that have

been fuzzed, or are not of interest in the queue, are skipped with a very high probability (99%). Even if no new seeds of interest are found, they are skipped with a very high probability (95% for fuzzed seeds and 75% for non-fuzzed seeds).

3. Motivation

In this Section, we present several examples to illustrate the current problems of DGF. We suggest four issues that DGF needs to solve.

3.1. Delay Effect

Most current DGFs are based on AFLGo. Taking being "fast" as a criterion, AFLGo is clearly not fast enough. In the exploration phase, distances are given the same amount of energy. There are many long-distance seeds in the seed queue and short-distance seeds are not immediately available for scheduling. Although some executed or uninteresting seeds at the front of the queue are skipped with high probability when new seeds are generated, they will still be executed one after another for new seeds that are not previously generated. Even if the execution reaches a seed with a small distance, it produces a new seed that is considered to be valuable. This new seed is still added to the end of the queue and waits for the other massive seeds to finish scheduling.

This leads to a "delay effect"—the new seeds with small distances generated by the input mutation are scheduled after the other seeds have been executed, resulting in further distance reduction. As shown in Figure 3a, if the next seed in the seed queue to be fed into the program is S1 with a distance of 10, S1 undergoes a series of mutations and successfully produces a new seed Sn + 1 with a distance of six. This becomes the seed with the minimum distance in the current queue. At this time, there are still many other seeds in the queue, which may be further away from the target distance point, or, due to the increased number of coverage of some edges, it joins the seed queue. At this point, they are less valuable than Sn + 1, in terms of both distance and coverage. Seeds continue to be scheduled as shown in Figure 3b. S2 generates seeds Sn + 2 with a distance of nine after mutation. In this way, Sn + 1 will only be scheduled after all previous seeds that may not be valuable have been executed. Seeds may continue to be produced that are not as valuable as Sn + 2, but are considered valuable to the seed queue. This results in a delayed effect for valuable seeds.

3.2. Path Ignored

There are often many paths to the target point. The necessary condition to trigger a crash may exist just on a certain path. Consider the use-after-free (UAF) vulnerability as an example. When the memory block is released, its corresponding pointer becomes a dangling pointer and does not point to any legal address. If it is used again after it is set to null, the program will crash.

In Figure 1, T1 corresponds to the release of the variable and T2 corresponds to the use of that variable. We have previously calculated that the distance 0.507 for input1 covering two target points is significantly smaller than the distance 0.622 for input2 covering only one target point. In this case, the fuzzer prefers to generate test cases covering multiple target points.

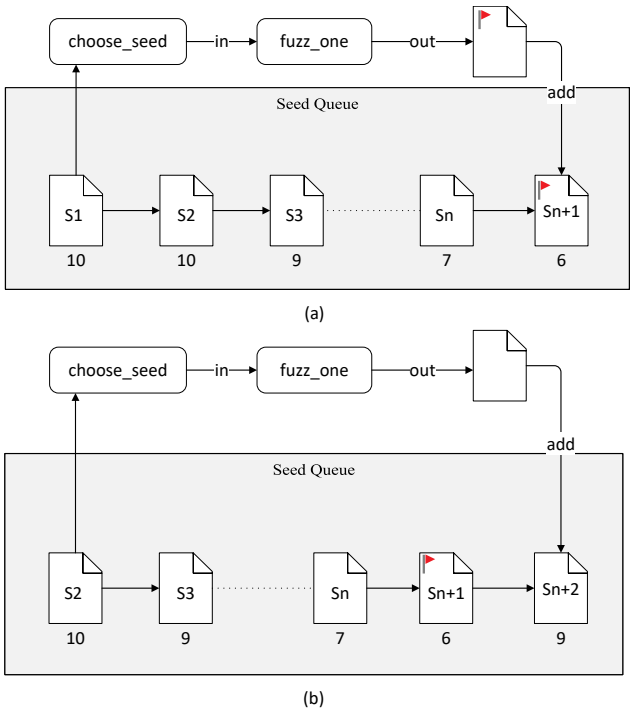

Figure 3. Delay effect of seed queue. (**a**)Valuable seeds Sn added to the queue. (**b**)Less valuable seed Sn+1 added to the queue.

However, for applications such as patch testing, we have no way of knowing for certain which paths are dangerous or how other target points should be set. In this case, as shown in Figure 4, the preconditioned free space statement that triggers the UAF vulnerability is on a long path that passes through more basic blocks. A crash does not occur when the short path is executed. At this point, for input1 via A-B-T to reach the target, the distance is $(2 + 1 + 0)/3 = 1$. The input2 that can trigger the vulnerability passes through A-B-C-D-T. The distance of input2 is $(2 + 1 + 2 + 1 + 0)/5 = 1.2$. In such a case, the long paths that pass through more basic blocks tend to be ignored because their distances tend to be larger. So, even for directed fuzzing, we want to reach the target both quickly and with as many paths as possible to the target point.

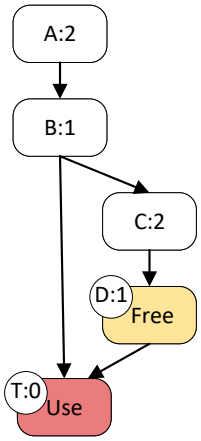

Figure 4. A case of crash triggering experiencing a long path.

AFLGo seeks to explore as many paths as possible in the exploration phase. While making it slower to reach the target point (as in the previous analysis), the timing of this phase also needs to be specified by the tester before performing the fuzzing. The testers do

not know how long this is set to be appropriate. For AFLGo, it has only one exploration phase, and the ability to reach the target point and trigger the vulnerability depends heavily on the seeds generated in the exploration phase. If the time is set to be too small, the fuzzer will not adequately accumulate outstanding seeds. Some inputs with long paths to the target point, even if they are found, will not get energy due to the annealing algorithm. The final test cases generated tend to be concentrated on certain paths. This will result in a situation where the paths are locally optimal. The execution is limited to short paths and it is easy to become stuck in a certain place for a long time due to certain complex check conditions. This was confirmed in a previous study [3]; within a certain time frame, the longer the exploration time was set to be, the more likely it was to trigger a crash in subsequent tests. If the time is set to be too large, seeds at larger distances will take a lot of time. The really valuable seeds are drowned in the sea of seeds and cannot be dispatched in time. This also results in performance degradation.

3.3. Unreachable Path

As mentioned in Section 3.2, users of the fuzzer often cannot set the exploration time well. This can result in other problems.

Considering Figure 5 as an example, assume that the vulnerability triggering requisite is at D on the long path. However, D is not set as the target point. After a long exploration, there are seeds in the queue for paths A-B-C-T and A-B-E-F-G-T to reach the target point. For AFLGo, if the A-D path is not discovered during the exploration phase, it will be difficult to execute the path again at a later stage. The result of mutation of the seeds after this is that path A-B is executed with high probability. Even if the path to A-D-E is occasionally generated by mutation, it will be greater in distance than A-B-E. Over time, it will gradually become submerged in a pile of seeds at a much smaller distance.

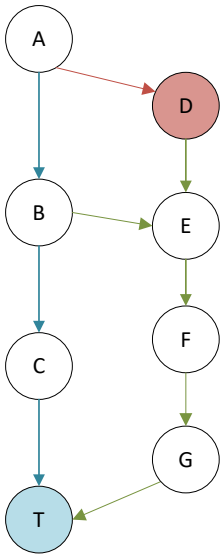

Figure 5. The case of possible missing vulnerabilities.

For the fuzzer, the A-D path is a path to the target point, but we cannot obtain the seeds for this path. Unlike the problem mentioned in Section 3.2, test cases for the route A-D path are unlikely to be generated because the time setting of the exploration phase is not reasonable.

Based on the above analysis, we pose the following questions.

Q1: For directed grey-box fuzzing, how can the target point be reached more quickly?

Q2: How can path diversity be better preserved and multi-path coverage be achieved using the directed grey-box fuzzing test?

Q3: For a seed that may be submerged in the queue with low energy, how can it be given a chance to be scheduled and when should this be considered?

Q4: How can speed and multi-path coverage be balanced? What strategy should be adopted?

4. Overview

The analysis above highlights four problems that need to be solved. These essentially concern the problem of (1) balancing the pursuit of speed to reach the target point directly, while (2) covering as many paths as possible that will reach the target point.

In this section, we present the general workflow for the method, as shown in Figure 6. SwitchFuzz is implemented based on AFLGo, which consists of two parts: a static analysis module and a fuzzing loop. In the static analysis module, we collect the call graph (CG) and control flow graph (CFG) of the program and insert the distance calculation module into the program at compile time. In the fuzzing loop, SwitchFuzz selects the appropriate seed queue for scheduling, saving it according to the current state, and continuously mutates and optimizes to generate seeds that are closer to the target point.

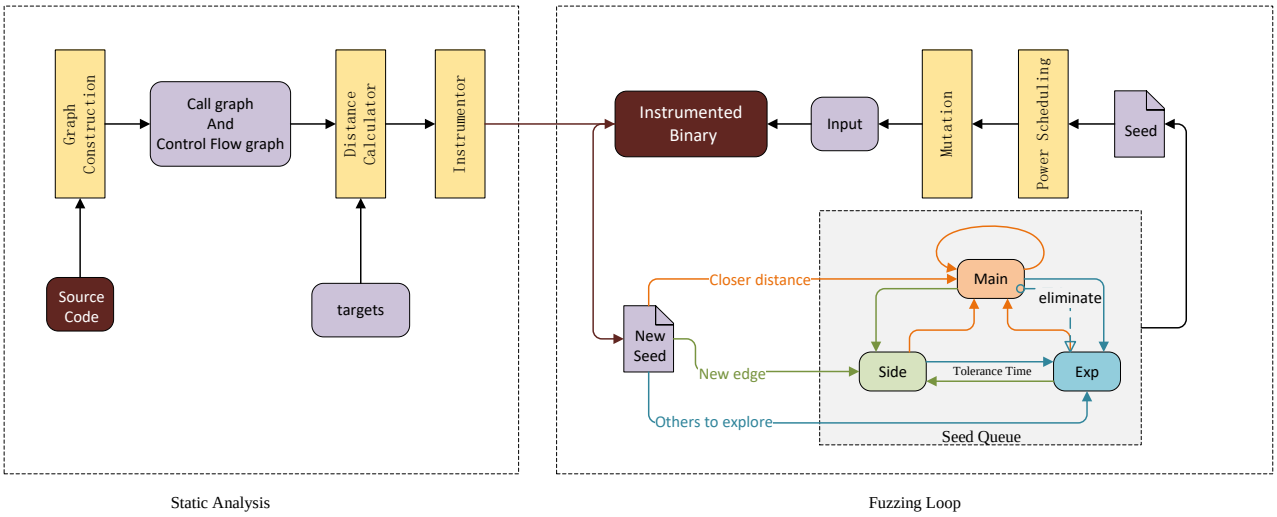

Figure 6. Approach overview of SwitchFuzz.

4.1. Static Analysis

The input to the static analysis module is the program source code and target points (e.g., code files and line numbers) and the output is a binary program with instrumentation. SwitchFuzz is distance-oriented, which requires calculating distances before the static analysis phase and staking them during the compilation phase. The AFLGo calculation scheme is followed in this part. The CG and CFG of the source code are extracted based on which function-level distances and basic block-level distances are obtained. The distance of each basic block from the target point is calculated and the distance calculation code is staked into the binary program at compile time. During dynamic time, the stubbed code performs the distance calculation to obtain the distance from the seed to the target point.

4.2. Fuzzing Loop

The fuzzing part is a circular process where the input is a binary program with instrumentation and an initial seed. The output is a seed file that can cause the program to crash or time-out. Algorithm 1 shows the workflow of SwitchFuzz in a fuzzing loop.

SwitchFuzz maintains three different seed queues. At the beginning of the fuzzing loop, it selects the appropriate queue for scheduling based on the current execution status and chooses the next seed from it. For the different seed queues, different skip algorithms are adopted (line 3 and line 4).

SwitchFuzz follows the power scheduling algorithm of AFLGo. In the early part of the run, each seed is given the same energy. As the run time gradually reaches a threshold, the seeds closer to the target point have higher energy (line 5). The difference is that

SwitchFuzz calculates the runtime independently for each queue, ensuring that more of the longer distance paths that had been overlooked are found when exploration is performed. For higher energy seeds, it will spend more time on mutation and testing (line 7).

Depending on the result of the execution, SwitchFuzz saves the seeds to a different queue. It is worth noting that the switching of seed queues is only for scheduling and usage. New seeds may still be added to the queue, even if they are not in the time slice of that queue (line 8 and line 9).

Algorithm 1: Fuzzing loop.

```
Input: S, the seed input
Output: S_c, the crash testcase
initialization;
while no timeout reached or abort-signal do
    queue = choose_queue();        //Our modifications
    s = find_next(S,queue);        //Our modifications
    p = calculate_energy(s)
    repeat p times:
        s' = mutate_inputs(s);
        result = execution(s');
        save_seed(s',result)       //Our modifications
end
```

5. Methodology/Design

In this section, we provide details of the scheme represented in Figure 6.

5.1. Different Seed Queues

In Section 3, we posed four questions. We hope to properly balance speed and coverage and to find the missed paths. Consider the following: In many games, the protagonist often starts out with a mission, which we generally call the main quest. In order to clear the game, players will continue to complete the main quest in order to progress the game. However, as the difficulty level increases, advancing the main line becomes difficult. Players often need to complete side-missions to unlock new equipment, which can serve as props to continue the main challenge. After clearing the game, players seek to discover new things in the game. They will continue to complete side-missions or explore other routes.

By analogy to fuzzing, we distinguish between "main quests", "side quests", and "exploration routes" through three queues. Each of them corresponds to quickly reaching the target point, exploring multiple paths and finding the overlooked "potentials". The three queues are named "Main", "Side", and "Exp". For them to achieve their respective functions, we impose a limit on the seeds added within each queue. Algorithm 2 shows the allocation of seeds.

5.1.1. Queue_Main

For queue_main, its task is to complete the "main task" of hitting the target point (Q1) as fast as possible. So, we store all seeds that are closer to the target point. Subsequent seeds generated with greater distance will not be added to this queue. However, there are other problems with this design. We mentioned the "delay effect" in Section 3.1, where the "short distance" previously found in queue_main becomes less short as time goes on. We need to constantly optimize this queue by limiting the number of seeds contained within it, as shown in Figure 7.

Algorithm 2: *save_seed*(): Add the seed to the corresponding queue.

```
Input: S, the input test case
Output: queue_main, queue_side, queue_exp
if newbits_in_map≡false then
    return;
end
if cur_distance < min_distance then
    if main_size > capacity then
        eliminate_add(s);
    end
    else
        add_to_queue(s, queue_main);
    end
end
else if SeedWithNewEdge(s) ≡ true then
    add_to_queue(s, queue_side);
end
else
    add_to_queue(s, queue_exp);
end
```

When there are closer seeds generated that exceed the capacity, it is necessary to consider removing the long-distance seeds from the queue, such as the blue seeds in Figure 7. When this seed queue is continuously scheduled, the process of executing deeper paths may become stuck due to certain complex check conditions or for other reasons. This leads to the possibility that the shortest path currently being executed is not the one that reaches the target the fastest. In addition, limiting mutation to the shortest distance seeds may not lead to new advances. We may need to look for help elsewhere, such as in the other two queues.

When the other queues are executed, if a seed with a shorter global distance is found, it will also be added to the first queue. Its execution path may no longer be the same path it was before. However, we are not concerned, because now this is the seed that is the shortest from the target. Later in the execution, the distance is not updated for a long time because the program may have reached the target point. The target point is hit multiple times, which causes the valuable seeds to be added to the queue_exp. This also means that queue_main's task is done and, therefore, it is no longer scheduled. Our design also addresses this issue.

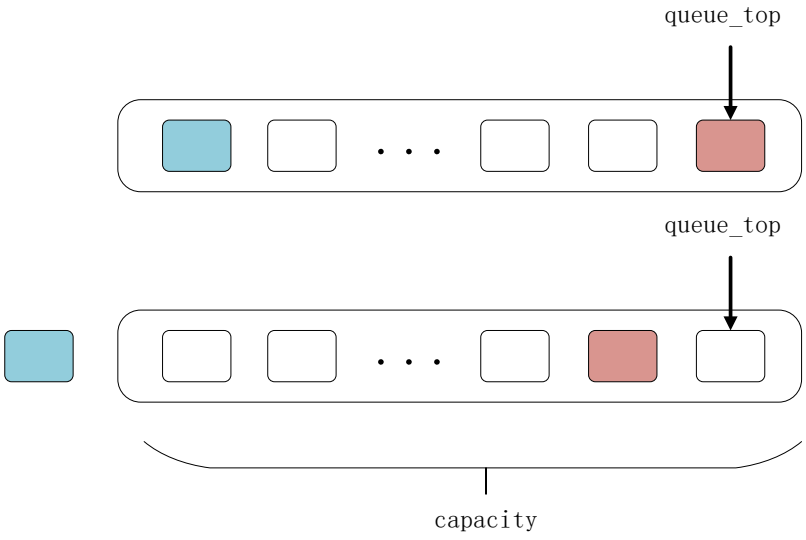

Figure 7. The case of a new seed added to queue_main.

5.1.2. Queue_Side

The job of queue_side is to complete the "side quest" that involves exploration of more paths (Q2). It holds the seeds for new edges that are found. To produce new coverage there are two cases: one is where a new basic block has been reached on the original path and is getting closer to the target point, such that the seed has been added to the main_queue; the other is where a new path has been found, which may not be as short as the previous path. Switching to invoke it when the target point is not reached can help resolve a stuck mainline, as discussed in Section 5.1.1.

After reaching the goal point, we assume that the necessary conditions exist to trigger a crash on other branches, or that the execution of other paths in the pass will be more likely to trigger a crash when the mainline fails to produce a crash for a long time. So, we choose to switch to queue_side to try to reach the target point via multiple paths to trigger a crash.

5.1.3. Queue_Exp

For queue_exp, the task is as its name suggests, that is, exploration (Q3). Queue_exp holds the other seeds that are not added to the first two queues. It should be clarified that we do not change the conditions under which the generated test cases are used as new seeds but only add a new classification rule. Giving too much energy to queue_side in the first stage will cause the guidance of the directed grey-box fuzzing to become similar to a coverage guidance mechanism, which is contrary to our original intention.

Therefore, we separate out another exploration queue to reach our destination directly. It does something similar to what AFLGo does in its exploration phase. We dispatch it only when we need it to discover new possibilities. This design enables accumulation of test cases in a fixed and reasonable period of time. It explores the seeds needed for the other two queues, keeping the seeds for shorter new paths and longer new covers.

5.2. Try Another Way—Switch Queue

It can readily be seen from the description in Section 5.1 that each of SwitchFuzz's three queues performs a different task at a different stage. When the current task is blocked for a long time, its other teammates replace it and solve the puzzle for it while finishing their own work. Then the question arises of under what circumstances the queue should be switched and to which queue (Q4). This is discussed below.

5.2.1. How to Switch

Let us consider the four questions posed in Section 3. First, we need to get to the target point faster. This task is undertaken by queue_main, so we will call queue_main first during execution. For queue_main, it executes the seeds whose distance is getting shorter. As it continues to execute, the distance to the target point will get smaller and smaller until it gets stuck or reaches the target point. With the queue_main execution, we can get to the target point quickly, which is what DGF expects. So when other queues in a fuzzing loop find a seed with a shorter distance and add it to queue_main, there is no doubt that we need to execute queue_main immediately to try to hit the target point (I).

When it finds a new seed that is closer during its own execution, which is what we want to see, we should continue to give the queue a chance to further reduce the distance (II).

When queue_main cannot make progress, it may be because it cannot continue advancing the current path. Then, we need to find another path to switch exploration. Alternatively, it might be because the target point has already been reached. Then, we need to try multiple paths to reach the goal point to check if there are vulnerabilities. At this point, to accomplish this task, we need to cut back to other queues (III).

For the other two queues, we switch based on a parameter called the tolerance time (in Section 5.2.2). If no new progress is achieved after this time, switch to the other queue (IV). We formulated the switch flowchart shown in Figure 8.

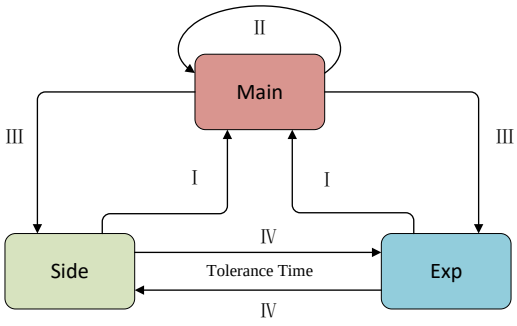

Figure 8. Switch flowchart.

5.2.2. When to Switch

As stated in Section 5.2.1, new seeds found in queue_main need to be dispatched immediately. Due to the limited number of seeds within it, even frequent scheduling does not result in significant time wastage. Each execution implies that a seed closer to the target is created, so we take the number of execution rounds n of the queue as the limit. If a closer seed is found in the process, the execution rounds are reset and the execution continues. If a closer seed is not found after n execution rounds, the queue is switched back to the previous one.

For the other two queues, we take a time-slice scheduling approach, similar to that of a time-sharing system—a time is assigned to each of the two queues, and it switches to the next queue when the execution time exceeds this time-slice. Unlike traditional time-slices, we do not use execution time as a criterion. Instead, we use the time since the last new coverage was discovered. We set a variable called "tolerance time" for each queue before it is scheduled. At the end of each fuzzing loop, the queue will be switched to the next one if the time since the last coverage is found to exceed the tolerance time. This design produces a queuethat is able to refresh its timings after new coverage is found, allowing it to continue execution. This also makes it possible to make a queue switch to try to complete other tasks or to get some help for this queue when we cannot obtain satisfactory results for a long time. Let us now consider how the tolerance time should be obtained. Equations (1)–(3) show how the tolerance time t_t is calculated.

In Equation (1), t_{ann} is the shortest distance since the last update. When there is no means of obtaining a shorter distance, we assume that the exploration is stuck halfway or has reached the target point. In either case, we need more time to execute queue_side. r_{ann} is a time factor, which is set by the user before execution. When there has been no distance update, it determines how long the tolerance time will be to increase to twice as much as that of the queue_exp.

$$c_{ann} = \begin{cases} 1 & queue\_exp \\ 2 - e^{-t_{ann} * r_{ann}} & queue\_side \end{cases} \quad (1)$$

We have a time factor denoted c_{ann}, but no time base. How to design a suitable time base then becomes a problem. Some programs resulting from bad code writing habits can easily trigger a crash. Even if it can trigger a crash occasionally in one queue, it may not be as fast as the other queues, analogous to squeezing out the last bit of toothpaste. For programs that have more difficulty triggering crashes, frequent queue switching is not very helpful in finding crashes, because they are not stuck halfway but arise because most parts of the program do not have vulnerability points. So we need to be able to adjust this time base through a dynamic process.

In Equation (2), t_c indicates the time base of a queue execution for subsequent calculation of the tolerance time and $count_{fd}$ denotes the number of execution rounds in which a unique crash is consecutively found. Each execution round loop includes three queue-switching schedules. Even though queue_exp may be skipped, we still consider that switching back to queue_main adds a round. $count_{nfd}$ indicates the number of consecutive

execution rounds where no unique crash is found. $count_c$ indicates the number of times that t_c is changed in the same direction (zoomed in or out). For example, the initial time base is n. When the execution finishes one round, a new unique crash still cannot be generated. At this point, we believe that it is not that the individual queues are not performing well and need to be switched frequently, but that it is harder to trigger crashes overall. Then, we let the time base expand to $2n$. The next expansion to $4n$ requires two consecutive rounds, the expansion to $8n$ requires four consecutive rounds, and so on. If it is scaled down, it will still only take one round. This allows t_c to remain within a range of the initial t_c without becoming too large or too small. Even though we still need to specify an initial time base t_c before the test as AFLGo does, it can be dynamically adjusted according to the execution status to make the tolerance time more reasonable.

$$t_c = \begin{cases} \frac{1}{2}t_c & count_{fd} = 2^{count_c} \\ 2t_c & count_{nfd} = 2^{count_c} \\ t_c & \text{else} \end{cases} \tag{2}$$

Finally, multiplying the time factor c_{ann} with the time base t_c gives the tolerance time, as shown in Equation (3).

$$t_t = c_{ann} \times t_c \tag{3}$$

5.3. Skip or Not—Find Next Seed

We limit the queue_main queue size and always keep the shortest distance seeds inside it. If we still use the AFLGo mechanism, there is a high chance that most of the seeds in it will be skipped because they have markers named "was_fuzzed", which we do not want. Therefore, we set a separate skip rule for this queue; Algorithm 3 shows the skip rule.

Algorithm 3: *find_next()*: Find next seed to be fuzzed.

```
Input: queue_cur, entry of the current queue
Output: s, the next seed to be fuzzed
while queue_cur do
    if queue_cur ≡ queue_main then
        if !queue_cur->was_fuzzed || !skip_p(MAIN_SKIP) then
            return queue_cur;
        end
    end
    else
        if !skip_p(ORIG) then
            return queue_cur;
        end
    end
    queue_cur = queue_cur->next;
end
```

For seed selection of other queues, we follow the rules of AFLGo. This rule has been introduced in Section 2.3.

In the queue_main queue are the seeds that we have filtered out to be extremely valuable. For seeds that are not fuzzed, we will definitely use them. For the fuzzed seeds, we still think they have potential. Most of the generated test cases are the result of non-deterministic variants, which have large randomness. We decided to give the queue a second chance, by skipping with a probability called $MAIN\_SKIP$. The setting of this parameter depends on the queue capacity. When the queue capacity is large, skipping each previous seed with a small probability will make the queue bloated. When the queue capacity is small, skipping with a large probability will make it difficult for the queue to be effective.

Considering Figure 7 as an example, we assume that n rounds will be executed each time we switch to queue_main. For the red seed just added to queue_main, as new seeds are added, its position in the queue is gradually moved to the end until it is removed. Each time, it will be skipped with some probability. If the execution probability is p and the queue capacity is c, the expected value of the number of executions is $E = n(p(c-1)+1)$.

$$p = \frac{E-n}{n(c-1)} \tag{4}$$

6. Results

6.1. Parametric Test

As far as we know, it is not easy to set appropriate parameters for different programs. In order to select appropriate parameters, we designed experiments to obtain the distribution law of the number of times the seed has been executed when the distance is shortened.

We insert a seed structure variable called *usage_count* for the record. This variable is incremented each time the seed is reused. When the distance gets shorter, the number of executions will be logged into an array. Figure 9 depicts the distribution of the number of times the current seed is executed when the distance is shortened after 4 h. The vertical coordinate represents the number of seeds and the horizontal coordinate represents the number of seeds reused (that is, *usage_count*). The dataset used is introduced in the follow-up formal experiments section.

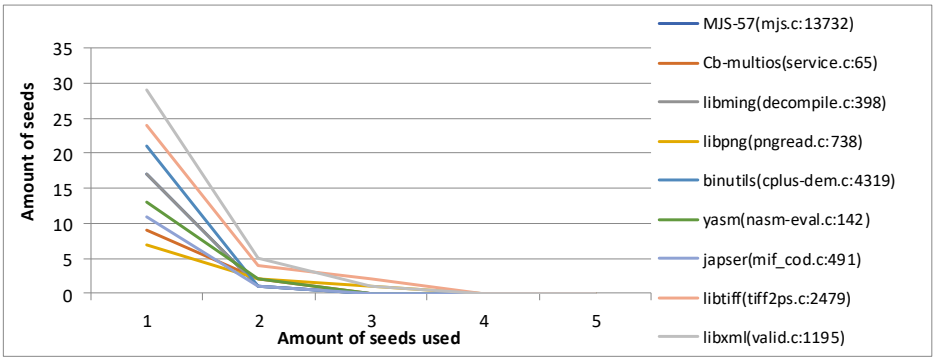

Figure 9. The distribution of the number of times the current seed is executed when the distance is shortened after 4 h.

When mutating a seed repeatedly used in the *queue_main* to get a shorter distance test case, most of the seeds are only reused once or twice; a very small fraction of the seeds show their value only after "three times". Based on this result, we believe that setting the expectation E of seed execution times to five is sufficient to bring out the value of seeds. We also set the skip probability to 0.5 in order to give more attention to the seeds that have just entered the queue. At this point, in Equation (4), n(c + 1) = 10. Obviously, the capacity c is an integer greater than one and the number of executions n is a positive integer. So c can be set to four or nine and n can be set to two or one. However, if the capacity is taken to be nine, the seed queue will be too bloated, causing the long-distance seeds in it to be dispatched multiple times. This will definitely cause the efficiency of this queue to decrease. So, finally, we set the capacity c to four and the number of executions n to two.

6.2. Crash Exposure Capability

We applied these strategies based on AFLGo to form our directed grey-box fuzzer called SwitchFuzz. To verify the effectiveness of the solution, we applied SwitchFuzz to real programs, such as binutils, mjs [4], and libming [5] to test it. Unfortunately, CAFL [6], Hawkeye [7], Lolly [8] and other DGF tools are not open-source. We evaluated the performance of SwitchFuzz against AFLGo according to the questions posed in Section 3.

The "time-to-exploitation" value of AFLGo was set to 45 min. To reduce randomness due to variability, all experimental groups were repeated eight times.

Our experiments ran on an Intel® Core™ i7-4790 CPU @ 3.60 GHZ, which included eight core processors and allocated four core processors to the experimental environment. The system version was Ubuntu 16.04.7 LTS. The memory was 16 GB in size. The original experimental dataset (including program version, initial input and target code lines) and the complete experimental results were saved at figshare [9]; the link is https://figshare.com/articles/dataset/SwitchFuzz/20227509 (accessed on 28 June 2022).

The main goal of DGF is to quickly trigger a crash at the target point for patch testing, vulnerability recurrence, and other purposes. So the most direct indicator is the ability to reproduce crashes. We tested this on different programs. For the same application, we set different targets as target points to trigger different parts of crashes.

GNU binitils [10] is a set of binary tools that contains several committed CVE vulnerabilities in test version 2.26. We specify the target by CVE description and crash trace, and restore the site that triggered the crash at the end to confirm if it is the specified bug. For vulnerability recurrence, we can know the entry points of individual functions exhaustively. Therefore, we stake all entry functions as targets at compile time for better guidance. In many cases, it is impractical to know all entry function points exhaustively. It is also impractical to use the entry points of functions that can reach multiple paths of the code block to be tested as targets for distance calculation. Therefore, in our experiments, we tested single and multiple targets separately when setting up the target points. The nT column and $1T$ column in the Table 1 represent the test results under the staking condition of multiple targets versus single targets.

Runs in the table denotes the number of times the corresponding vulnerability is reproduced in eight tests with 8 h as a round of testing. TTE (time to exposure) denotes the time from the start of the fuzzing to the first trigger of the specified error. Factor is the ratio of the time used by AFLGo over SwitchFuzz.

In addition, we also compared other realistic programs; the table lists relevant information such as program names and vulnerability trigger points. The information for the multi-target settings can be found in the original dataset mentioned above. Due to space limitations, runs are denoted by Runs in the table.

As shown in Tables 1 and 2, both TTE and Runs are overall better than 1T in nT. This is even more evident for some of the harder-to-trigger vulnerabilities, such as libxml valid.c:1195. Benefiting from SwitchFuzz's ability to try to explore multiple paths when testing conditions are poor, SwitchFuzz does not rely too heavily on intermediate key-entry target settings. It still has a high probability of triggering the specified vulnerability in 1T. As in the testing of yasm 1.3.0 nsam-parse.c:1349, SwitchFuzz still exposed the vulnerability seven times. In contrast, AFLGo plummeted from eight to four times. In terms of test time, SwitchFuzz triggered vulnerabilities 1.47 times longer than AFLGo in nT, and 1.37 times longer in 1T. It is worth stating that we chose to average the time it took for the vulnerability to be triggered in eight trials. If the vulnerability is not triggered within 8 h, we do not include this result in the calculation of TTE. However, the actual time it triggers the vulnerability will be greater than 8 h. This means that with different runs, the actual time efficiency gain will be much larger than the current ratio. This is why SwitchFuzz does not appear to be as efficient as AFLGo for the CVE-2016-6131 experimental group in nT. SwitchFuzz also shows better performance for some of the easier triggered vulnerabilities. libtiff 0.4.7 tiff2ps.c:2479 is 2.23× faster in 1T and libxml valid.c:1181 is 2.03× more efficient in nT.

Table 1. Performance of AFLGO vs. Switchfuzz in terms of crash recurrence capability on binutils.

CVE	Tool	nT			1T		
		Runs	TTE(s)	Factor	Runs	TTE(s)	Factor
2016-4487	SwitchFuzz	8	185	-	8	276	-
	AFLGo	8	354	1.91	8	443	1.61
2016-4488 2016-4489	SwitchFuzz	8	108	-	8	272	-
	AFLGo	8	181	1.68	8	424	1.56
2016-4490	SwitchFuzz	8	89	-	8	114	-
	AFLGo	8	133	1.49	8	133	1.17
2016-4491	SwitchFuzz	2	14,355	-	2	25,728	-
	AFLGo	1	14,032	1.02	0	-	-
2016-4492	SwitchFuzz	8	740	-	8	1284	-
	AFLGo	8	1238	1.67	8	3267	2.54
2016-4493 2016-6131	SwitchFuzz	4	22,144	-	2	23,264	-
	AFLGo	2	19,630	0.88	0	-	-

Table 2. Performance of AFLGO vs. Switchfuzz in terms of crash recurrence capability on 11 targets in 7 real-world programs.

Project	Tatget	Tool	nT			1T		
			Runs	TTE(s)	Factor	Runs	TTE(s)	Factor
MJS 1.21	mjs.c:13732	SwitchFuzz	6	12,520	-	4	20,221	-
		AFLGo	4	14,234	1.14	1	21,175	1.05
cb-multios[11]	service.c:65	SwitchFuzz	8	21	-	8	17	-
		AFLGo	8	28	1.33	8	24	1.41
libming 0.4.8	decompile.c:398	SwitchFuzz	7	6798	-	8	9365	-
		AFLGo	6	8144	1.2	5	16,896	1.8
libming 0.4.8	decompile.c:349	SwitchFuzz	8	13,073	-	5	16,503	-
		AFLGo	7	21,004	1.61	3	19,083	1.16
yasm 1.3.0	nasm-parse.c:1349	SwitchFuzz	8	9543	-	7	16,235	-
		AFLGo	8	14,007	1.47	4	22,305	1.37
yasam 1.3.0	nasm-eval.c:142	SwitchFuzz	6	22,374	-	7	22,572	-
		AFLGo	4	22,766	1.02	2	26,542	1.18
yasam 1.3.0	intnum.c:415	SwitchFuzz	7	14,698	-	7	17,915	-
		AFLGo	5	21,634	1.47	3	24,040	1.34
jasper[12] 1.900.1	mif_cod.c:491	SwitchFuzz	8	83	-	8	114	-
		AFLGo	8	103	1.24	8	140	1.22
libtiff 0.4.7	tiff2ps.c:2479	SwitchFuzz	8	1353	-	8	1512	-
		AFLGo	8	1999	1.3	8	3376	2.23
libxml 20902	valid.c:1181	SwitchFuzz	8	255	-	8	420	-
		AFLGo	8	518	2.03	8	705	1.68
libxml 20902	Valid.c:1195	SwitchFuzz	7	15,092	-	5	21,516	-
		AFLGo	4	16,995	1.13	3	24,178	1.12

6.3. Edges Coverage

As we showed in the experimental analysis above, we obtained an improvement in the probability of triggering the vulnerability by re-exploring the queue_side. For hard-to-reach target points, it is appropriate to cover more edges, which helps us to get closer to the target point. To further verify the performance of SwitchFuzz in this respect, we selected some programs that took a long time to reach as a test dataset, recording the shortest distance and the number of edges found. Each set of experiments was set for 4 h and was performed under 1T conditions. The targets set and the results obtained are shown in Table 3.

Table 3. Edges and minimum distance found by SwitchFuzz and AFLGo.

ID	Project	Fuzzer	Edges	Minium Distance
#1	jasper (mif_cod.c:491)	SwitchFuzz	23	488.27
		AFLGo	21	492.15
#2	MJS 1.21 (mjs.c:13732)	SwitchFuzz	262	2928.29
		AFLGo	250	2928.29
#3	MJS 1.21 (mjs.c:4908)	SwitchFuzz	277	2770.78
		AFLGo	275	2770.78
#4	cb-multios (service.c:65)	SwitchFuzz	6	371.92
		AFLGo	5	705.05
#5	libpng[13] 1.2.56 (pngread.c:738)	SwitchFuzz	85	821.09
		AFLGo	82	821.09
#6	libming 0.4.8 (decompile.c:349)	SwitchFuzz	708	1423.73
		AFLGo	653	1551.32

For the number of edges found, SwitchFuzz can harvest more edges when re-exploring. For some of the harder-to-reach edges, AFLGo gives up exploring after the exploration time. This causes its distance from the target to stop shortening as well. SwitchFuzz tries to switch the queue to find a new edge to get closer to the target when the distance is no longer shortened. Note that, since the multi-target distance calculation mechanism of AFLGo is followed, a non-zero distance does not mean that the seed does not reach the target point to trigger a crash.

In addition, finding more edges is also meaningful for distance reduction, which was most evident in testcase #4. SwitchFuzz found only one more edge than AFLGo, but the distance shortened from 705.05 to 371.92. We also found in our experiments that, for AFLGo in testcase #1 and #4, the minimum distance stayed at a certain value for a long time and failed to discover new edges or crashes for a long time. SwitchFuzz was able to achieve the distance reduction faster by exploration of queue_side after a certain time stay. This is also a good illustration of the role of queue_side, which is to explore more paths to solve the "stuck" situation when the target cannot be approached or crashes are triggered.

7. Related Work

Related research is described below.

7.1. Grey Box Fuzzing

Compared to white-box fuzzing, grey-box fuzzing requires only lightweight program analysis to obtain the necessary program state information during the fuzzing process. AFL is a typical grey-box fuzzing tool that finds vulnerabilities by recording the code coverage of input samples and adjusting the input samples to increase the code coverage. Since it can be readily extended, many grey-box fuzzing studies have been conducted based on AFL.

In recent years, there have been many studies based on different aspects or components of grey-box fuzzing. We do not exhaustively cover all aspects, but only those that are relevant to our study.

7.1.1. Seed Generation

A proper initial seed can improve fuzzing efficiency to a great extent. For some objects to be tested that require fixed format input, if the corresponding seed input is not provided, fuzzers will spend a huge amount of time on finding valid test cases since the exploration space for mutation will be endless. For some more-difficult-to-pass branches, the fuzzing efficiency can be greatly improved by an effective new seed generation strategy. There are three research perspectives on this, as described below:

- **Combined with deep learning.** This approach starts from the legitimate input to the program, generates a model by learning the structure of the input extensively, and then uses the model to continuously generate the input. Examples include Learn & Fuzz [14], GANFuzz [15] and Neuzz [16].
- **Assisted with symbolic execution.** Fuzzing using symbolic execution is generally known as hybrid fuzzing. Some checks are difficult to pass, such as array checks during the fuzzing process, causing the fuzzer to spend a lot of energy trying to mutate to generate the eligible input. When a branch is difficult to pass, symbolic execution is used to solve it, generating a new seed and then passing it to fuzzing to pass the branch. Examples include QSYM [17], DigFuzz [18] and HFL [19].
- **Based on static analysis and dynamic analysis.** This approach uses the technique of program analysis to determine what kind of input the program accepts and then guides the generation of test cases. Examples include FANS [20] and T-Fuzz [21].

7.1.2. Seed Selection

In each iteration of the genetic algorithm, the seeds for the next step of fuzzing are selected from the current seed pool. The testing efficiency of selecting different seeds varies, so some methods have been developed from the perspective of seed selection.

For example, AFLFast [22] improves the seed selection strategy by prioritizing seeds that have been less frequently selected previously or that can reach paths that have been less frequently tested previously. Related examples include Vuzzer [23], QTEP [24] and CollAFL [25].

7.1.3. Mutation Strategy

In a fuzzing loop, fuzzers try to discover more program states or crashes and generate new seeds by mutating and testing the seed inputs. For fuzzing, the space for mutation is almost infinite. Choosing a suitable mutation strategy also has a great impact on the efficiency of fuzzing.

- **Based on optimization algorithms or deep learning models.** Guiding the mutation through optimization algorithms or deep learning models enables the fuzzing process to obtain efficient and usable test cases. For example, MOPT [26] selects the optimal variation strategy using the particle swarm optimization algorithm. ILF [27] uses symbolic execution to generate data as training data and AI models to guide the mutation.
- **Based on program analysis**. These types of studies usually combine methods such as taint analysis, symbolic execution, and gradient descent. After analyzing the program, it is determined which bytes should be mutated and what kind of mutation should be performed. Examples include Vuzzer, ProFuzzer [28] and GreyOne [29].

7.2. Directed Grey Box Fuzzing

Traditional grey box fuzzing is guided, based on code coverage. In the case of AFL, when a test case finds an edge that has not been covered, the input is used as a new seed to

continue the mutation and testing it involves trying to traverse from as many code spaces as possible.

The goal of directed grey-box fuzzing is, instead, to reach a predefined goal as soon as possible and to trigger a corresponding crash. Unlike directed symbolic execution, DGF transforms the problem of solving the arrival path into an optimization problem—the algorithm is used to continuously optimize the generation of inputs closer to the target point. Directed grey-box fuzzers, with AFLGo as an example, require static analysis of the program in the preliminary stage to obtain the function call graph and control flow graph. A distance calculation module is inserted into the target program during the compilation stage of the program and the optimization algorithm is designed during the fuzzing process based on the distance, so that the seeds close to the target can obtain more energy.

Hawkeye adopts a similar approach for energy scheduling [7]. Hawkeye introduces the covered function similarity. Seeds covering more functions in the "expected traces" will be given more chances to be mutated to reach the targets, which means that the long path receives more execution opportunities. It uses three queues with different priorities to ensure that valuable seeds can be scheduled first. There are two possible scenarios for discovering a new edge—either getting closer to the goal point on a path or finding a new path. Hawkeye jumbles both types of seeds into a top priority queue; then, what is processed in that queue may be multiple paths. It is significantly less efficient for DGFs that seek to reach the goal point faster. In addition, Hawkeye's seeds are no longer moved out after being placed in the queue. Some seeds that were once considered valuable will gradually become less valuable as more valuable seeds will subsequently be generated one after another. The problems mentioned in Section 3.2 remain. A similar approach to Hawkeye is Guided-Fuzz [30], which uses a more fine-grained variation strategy than Hawkeye. In addition, it computes priority in terms of coverage and distance for queue scheduling. Since it has almost the same improvement angle as Hawkeye, it also suffers from the path-ignoring problem.

It was mentioned earlier that it is very difficult to let users set a reasonable time for the exploration phase using different test software. RDFuzz proposes a frequency-guided strategy—counting the branch coverage, allocating energy and choosing the appropriate mutation strategy based on the frequency and distance of the code blocks executed by the seed. Such a strategy can improve the likelihood that the two phases fall into path local optimality due to unreasonable time allocation. However, this does not solve the problem of wasted overhead causing time setting to be too long in the exploration phase or the problem of not accumulating test cases for some specific paths with too short time setting.

In some cases, the target point has a sequential structure (e.g., UAF vulnerabilities) and test cases that do not reach the target point sequentially cannot trigger the vulnerability. In addition, there may be data condition constraints (e.g., buffer overflow) at some target points, which cause some test cases to fail to trigger vulnerabilities even if they reach the target point. CAFL proposes constraint-guided directed grey-box fuzzing (CDGF) to guide the seeds to satisfy a sequence of constraints. CAFL takes into account both control and data flows and defines the constraint distance, which is the sum of the target location distance and the data condition distance. CAFL considers how to give the seeds the right distance. SwitchFuzz considers how to use the existing distance of seeds for reasonable scheduling. So, SwitchFuzz's strategy can still be used on CDGF. Unfortunately, CAFL is not open-source and we cannot apply our strategy to CAFL at the moment. In addition, CAFL still produces the delay effect mentioned in Section 3.1. CAFL also requires exhaustive vulnerability information to complete the constraint template and cannot be used easily for code block testing.

AFLGo places much of the program analysis in the instrumentation to improve runtime efficiency. Lolly [8] was proposed as a sequence-coverage directed fuzzing (SCDF) method, a lightweight directed fuzzing technique. Given a sequence of target statements for a program, Lolly assigns energy by computing the "sequence coverage" of the seeds, with the aim of generating inputs that reach the statements in each sequence in order. It

reduces the overhead of instrumentation in AFLGo and also solves the problem that the target points have a sequential structure. Since sequence coverage utilizes the computation of the longest common subsequence, the basic blocks will be duplicated under program structures, such as loops, which will have some impact on Lolly. Compared with SwitchFuzz, Lolly still does not effectively solve the problem mentioned in Section 3.3 that the exploration phase is set too short and seeds with potential are not generated or scheduled under the single-goal test condition.

There are also what we consider to be "different" directed fuzzers. Their target points are not specified by the user, but are determined by the fuzzer. This type of fuzzer will determine the possible points of vulnerability in the program through analysis and then set these code sections as target points for directed fuzzing. For example, ParmeSan [31] utilizes many sanitizers with different detection functions that come with compilers, such as LLVM, to direct the fuzzers to specific classes of bugs. V-Fuzz [32] and Suzzer [33], on the other hand, combine a static analysis approach that uses deep learning models to predict program vulnerabilities and exposes the location of possible vulnerability targets for directed fuzzers. This type of fuzzer is different from the previously mentioned DGFs (including SwitchFuzz) in terms of application scenarios. Most of the traditional DGF application scenarios are patch testing, vulnerability recurrence or PoC generation. In such scenarios, their goal is only to trigger bugs at specified locations and it is not of concern whether there are vulnerabilities in other code fragments. Such fuzzers prefer to test the whole program, directed towards finding vulnerabilities more efficiently and comprehensively.

There has also been some research on fuzzing aids, such as FuzzGuard [34]. These use deep learning methods to help the directed gray-box fuzz tester filter unreachable inputs before seeding execution. This is very helpful for performance.

8. Conclusions

In this paper, we analyze current targeted gray-box fuzzing tools represented by AFLGo. Seed selection has a delaying effect. The valuable seeds generated will be scheduled only after the previous seeds are executed. In addition, directed fuzzing tends to ignore the impact of coverage on testing efficiency, leading to the problem of local optimality of paths without accumulating sufficient test cases with value. In this paper, we try to balance speed and coverage and conduct multi-path exploration where appropriate to better trigger vulnerabilities.

We apply the optimization scheme to SwitchFuzz. Experimentally, SwitchFuzz triggers crashes faster on easier-to-reach targets and has a higher probability of triggering crashes on harder-to-reach targets. SwitchFuzz performs better when a single target is set for testing without there being detailed information about the vulnerability.

Author Contributions: Conceptualization, P.J.; data curation, H.L.; formal analysis, Z.H.; funding acquisition, P.J.; investigation, Y.L.; methodology, Z.H.; software, Z.H.; supervision, Y.F.; validation, H.L.; visualization, Z.H.; writing—original draft, Z.H.; writing—review and editing, Z.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Key R&D projects of China OF FUNDER grant number 2021YFB3101803.

Institutional Review Board Statement: Not appliable.

Informed Consent Statement: Not appliable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zalewski, M. *American Fuzzy Lop (AFL) Fuzzer*, 2014 ed.; 2014. Available online: http://lcamtuf.coredump.cx/afl/technical_details.txt (accessed on 28 June 2022).
2. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344.
3. Ye, J.; Li, R.; Zhang, B. RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation. *Math. Probl. Eng.* **2020**, *2020*, 7698916. [CrossRef]
4. Software, C. mjs. 2016. Available online: https://github.com/cesanta/mjs (accessed on 28 June 2022).
5. libming. Libming. 2008. Available online: http://www.libming.org/ (accessed on 28 June 2022).
6. Lee, G.; Shim, W.; Lee, B. Constraint-guided directed greybox fuzzing. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 3559–3576.
7. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108.
8. Liang, H.; Zhang, Y.; Yu, Y.; Xie, Z.; Jiang, L. Sequence coverage directed greybox fuzzing. In Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE Computer Society, Montreal, QC, Canada, 25–26 May 2019; pp. 249–259.
9. Science, D. Figshare. 2011. Available online: figshare.com/ (accessed on 28 June 2022).
10. Binutils, G. GNU Binutils. 1990. Available online: https://www.gnu.org/software/binutils/ (accessed on 28 June 2022).
11. trailofbits. cb-multios. 2016. Available online: https://github.com/trailofbits/cb-multios (accessed on 28 June 2022).
12. Adams, M. Jasper. 2006. Available online: https://ece.engr.uvic.ca/~frodo/jasper/#doc (accessed on 28 June 2022).
13. Schalnat, G.E. Libpng. 1995. Available online: https://github.com/glennrp/libpng (accessed on 28 June 2022).
14. Godefroid, P.; Peleg, H.; Singh, R. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Champaign, IL, USA, 30 Octobe–3 November 2017; pp. 50–59.
15. Hu, Z.; Shi, J.; Huang, Y.; Xiong, J.; Bu, X. GANFuzz: A GAN-based industrial network protocol fuzzing framework. In Proceedings of the 15th ACM International Conference on Computing Frontiers, Ischia, Italy, 8–10 May 2018; pp. 138–145.
16. She, D.; Pei, K.; Epstein, D.; Yang, J.; Ray, B.; Jana, S. Neuzz: Efficient fuzzing with neural program smoothing. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), Francisco, CA, USA, 20–22 May 2019; pp. 803–817.
17. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
18. Zhao, L.; Duan, Y.; Yin, H.; Xuan, J. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019.
19. Kim, K.; Jeong, D.R.; Kim, C.H.; Jang, Y.; Shin, I.; Lee, B. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
20. Liu, B.; Zhang, C.; Gong, G.; Zeng, Y.; Ruan, H.; Zhuge, J. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 307–323.
21. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by program transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 697–710.
22. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
23. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.
24. Wang, S.; Nam, J.; Tan, L. QTEP: Quality-aware test case prioritization. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 523–534.
25. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), Francisco, CA, USA, 21–23 May 2018; pp. 679–696.
26. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimized mutation scheduling for fuzzers. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 1949–1966.
27. He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 531–548.
28. You, W.; Wang, X.; Ma, S.; Huang, J.; Zhang, X.; Wang, X.; Liang, B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2019; pp. 769–786.
29. Gan, S.; Zhang, C.; Chen, P.; Zhao, B.; Qin, X.; Wu, D.; Chen, Z. GREYONE: Data Flow Sensitive Fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2577–2594.

30. Lin, P.; Zhou, Z.; Zhang, S. An Improved Directed Grey-box Fuzzer. In Proceedings of the 2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC), Chongqing, China, 17–19 June 2020; Volume 9, pp. 1497–1502.
31. Österlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. ParmeSan: Sanitizer-guided Greybox Fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2289–2306.
32. Li, Y.; Ji, S.; Lv, C.; Chen, Y.; Chen, J.; Gu, Q.; Wu, C. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv* **2019**, arXiv:1901.01142.
33. Data, B. Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning. In Proceedings of the Information Security and Cryptology: 15th International Conference, Inscrypt 2019, Revised Selected Papers, Nanjing, China, 6–8 December 2019; Volume 12020, p. 134.
34. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2255–2269.