Critical Code Guided Directed Greybox Fuzzing for Commits

Yi Xiang[1,2], Xuhong Zhang[1,3,✉], Peiyu Liu[1,2], Shouling Ji[1], Xiao Xiao[1], Hong Liang[1], Jiacheng Xu[1], and Wenhai Wang[1,2,✉]

[1]Zhejiang University, [2]Zhejiang University NGICS Platform, [3]Jianghuai Advance Technology Center
E-mails: {xiangyi0406, zhangxuhong, liupeiyu, sji, xiao_xiao, hongliang, stitch, zdzzlab}@zju.edu.cn

Abstract

Newly submitted commits are prone to introducing vulnerabilities into programs. As a promising countermeasure, directed greybox fuzzers can be employed to test commit changes by designating the commit change sites as targets. However, existing directed fuzzers primarily focus on reaching a single target and neglect the diverse exploration of the additional affected code. As a result, they may overlook bugs that crash at a distant site from the change site and lack directness in multi-target scenarios, which are both very common in the context of commit testing.

In this paper, we propose WAFLGo, a direct greybox fuzzer, to effectively discover vulnerabilities introduced by commits. WAFLGo employs a novel critical code guided input generation strategy to thoroughly explore the affected code. Specifically, we identify two types of critical code: path-prefix code and data-suffix code. The critical code first guides the input generation to gradually and incrementally reach the change sites. Then while maintaining the reachability of the critical code, the input generation strategy further encourages the diversity of the generated inputs in exploring the affected code. Additionally, WAFLGo introduces a lightweight multi-target distance metric for directness and thorough examination of all change sites. We implement WAFLGo and evaluate it with 30 real-world bugs introduced by commits. Compared to eight state-of-the-art tools, WAFLGo achieves an average speedup of 10.3×. Furthermore, WAFLGo discovers seven new vulnerabilities including four CVEs while testing the most recent 50 commits of real-world software, including libtiff, fig2dev, and libming, etc.

1 Introduction

Software vulnerabilities can pose a significant security threat, making security testing on programs crucial. Within the thousands of lines of code that a program comprises, newly introduced code through commits for functional upgrades and bug patching can be critical, as such code has the potential to introduce new vulnerabilities to a program. For instance, recent studies have shown that nearly four in five bug reports in OSSFuzz [4] were regression bugs, indicating that the bug was introduced by a historical commit rather than the original codebase [39]. This emphasizes the need to comprehensively test commits, helping discover vulnerabilities introduced through code modifications early on.

While traditional coverage-based fuzzing [15, 21, 24–26] has proven to be highly effective in discovering security vulnerabilities in real-world software systems, it lacks efficiency in identifying vulnerabilities introduced through commits. This is because coverage-based fuzzing treats all parts of the code equally and does not specifically target the commit changes. By contrast, directed fuzzing [5, 14, 19, 27, 30, 38] is more suitable for commit testing scenarios as it focuses on testing specific parts of a program. Intuitively, one can guide the fuzzer to prioritize testing the new code and its affected areas by designating the commit change sites as targets.

However, in the specific context of commit testing, existing directed greybox fuzzers (DGFs) still encounter several problems. First, *existing DGFs primarily focus on quickly reaching the target (change site) while not guiding the fuzzer to thoroughly test the affected code.* Consequently, these fuzzers may fail to discover newly introduced vulnerabilities where the crash site differs from the change site, as they may not trigger crashes or even reach the crash site. In particular, the complexity of a program implies that a single modified line in a commit can significantly impact the entire program. For example, if a commit modifies the index of an array, all the code that uses that index to access the array is affected. As a result, an out-of-bounds vulnerability may occur at a location far away from the commit change site. By analyzing the issues of real-world programs on GitHub, we observe that over *half* of the crash sites associated with a bug introduced by a commit are different from the commit change site (refer to Table 1 "DiffLoc" for details). Moreover, existing techniques

✉Xuhong Zhang and Wenhai Wang are the corresponding authors.

even introduce counteractions to the affected code exploration. For instance, AFLGo [5] prioritizes inputs with shorter distances, resulting in the generation of inputs that follow similar paths and thus reduce the input diversity. Similarly, SelectFuzz [23] only considers and instruments the relevant code before the target, thus lacking important feedback to generate high-quality input after reaching the target. Thus, we need a new technique to discover commit introduced bugs that not only reach the change site but also thoroughly test the actually affected code.

Second, existing DGFs struggle to effectively handle the multi-targets problem, which is common in commit testing where a single commit can introduce multiple change sites. When testing a commit, it is crucial to comprehensively test every change site and its affected code. However, as the number of targets increases, existing DGFs experience a decrease in directness. In the worst case, their effectiveness may degrade to coverage-based fuzzing, lacking targeted guidance. The DGFs derived from AFLGo [5, 7, 11, 17] aggregate all target distances using the harmonic mean, which will overlook certain targets and compromise individual directness. On the other hand, DGFs based on path pruning [14, 23] aggregate the reachability analysis for all targets, which significantly decreases the pruned path ratio for each individual target. While there are some multi-target directed fuzzers like Parmesan [27] and FishFuzz [38], they primarily focus on sanitizer-marked targets, which often involve a large number of unrelated targets, unlike the commit targets that are interconnected.

To overcome the aforementioned problems, this work aims to propose an efficient directed greybox fuzzing for commit testing. There are two primary challenges: *1) How to quickly and thoroughly test the affected code?* Like existing works, we should first efficiently reach the change site (target). Once the target is reached, we have to maintain the reachability, and then generate diverse inputs to explore different program states of the affected code. *2) How to handle multiple site changes in a smart and lightweight manner?* An intuitive approach is to test each change site separately. However, this becomes impractical when a commit has numerous change sites. Moreover, this approach neglects important information, such as the fact that change sites within a commit often share the same context (e.g., being in the same function), resulting in a delay in bug discovery and inefficient utilization of resources.

To address the first challenge, we propose a novel critical code guided input generation strategy. The core design includes a critical code identification method, a target edge selection, and a mutation masking mechanism. Our key insight here is to generate inputs that explore diverse program states while preserving the execution of the critical code. Specifically, we identify two types of critical code: *path-prefix* code and *data-suffix* code. The path-prefix code refers to the control-flow prefix that can affect the change site, and the data-suffix code represents the code that has data dependency with the critical variables used in the change site. When mutating a seed, we check if it executes any critical code. If critical code is identified, we generate a mutation mask for the seed and apply fine-grained mutation to maintain its execution in the generated input. Otherwise, we resort to coarse-grained mutation to enhance the diversity of the seeds. Additionally, during the fuzzing process, we select the target edge based on whether the seeds reach the target. If the seed does not reach the target, we select the "closest" edge to the target among the executed path-prefix code of the seeds as the target edge. By mutating only the input bytes that do not deviate from the target edge, we gradually generate inputs that are closer to the target. If the seed already reaches the target, we select the rarest executed edge from both the path-prefix and data-suffix as the target edge for comprehensive testing of the affected code. The rare path-prefix indicates new paths toward the target, and the rare data-suffix indicates new data values that can impact the affected code. This approach allows our fuzzer to explore diverse program states that reach the target and thoroughly test the affected code.

To address the second challenge, we propose a new lightweight distance metric that maintains the directness of each target. Furthermore, we adjust the input generation strategy to optimize the exploration of the affected code. Specifically, we compute and record the distance for each target individually. To ensure that each target is guided effectively and receives sufficient testing attention, we only consider the distance of the rarest executed target when calculating the distance for an input. As for generating inputs to explore the affected code, we identify the critical code separately for each target and apply target edge masking accordingly. In addition, we simplify the complexity of testing targets by grouping them based on their common preconditions. Specifically, we employ the dominator tree concept to merge change sites within the same function into a merged target. These approaches allow our fuzzer to sufficiently test all the change sites.

In order to fairly compare our design with existing state-of-the-art fuzzers, we construct a benchmark dataset from the GitHub repository. This dataset consists of 30 real-world bugs from eight widely used C programs that have been tested in prior works [5, 11, 14, 23]. For each bug we manually identify its bug-inducing commit. In our evaluation, we compare the performance of WAFLGo with three categories of fuzzers. The first category includes directed fuzzers such as AFLGo [5]. The second category consists of coverage-based fuzzers like AFL [1]. The third category comprises regression fuzzers, specifically AFLChurn [39]. The experimental results demonstrate the superiority of WAFLGo compared to the other fuzzers. In general, WAFLGo achieves an average speedup of 10.3×. Besides, WAFLGo outperforms other fuzzers by successfully reproducing the highest number of bugs. Furthermore, we apply WAFLGo to test the newly submitted commits in real-world projects, including libtiff,

fig2dev, and libming, and successfully discover seven new bugs, including four CVEs. This real-world application of WAFLGo further validates its effectiveness in identifying bugs in newly committed code.

In summary, this paper makes the following contributions:

- We highlight the significance of thoroughly fuzzing the affected code in the context of commit testing, as solely reaching the commit change sites is insufficient for discovering newly introduced bugs.
- We propose a commit directed greybox fuzzer that utilizes a novel critical code guided input generation strategy and a new distance metric. These designs enable the generation of diverse inputs for thorough testing of all the commit change sites and their affected code, facilitating the discovery of newly introduced bugs.
- We implement a prototype of WAFLGo and make the source code available for future research at https://github.com/NESA-Lab/WAFLGo.
- We construct a dataset with 30 real-world bugs, and manually identify their bug-inducing commits (BICs). On the dataset, we evaluate WAFLGo through extensive experiments, and compare it with existing fuzzers. The comparison with existing fuzzers demonstrates its superiority. With WAFLGo, we successfully detect seven new vulnerabilities, including four CVEs.

2 Background

2.1 Greybox Fuzzing

Greybox fuzzing is an automated technique used to discover vulnerabilities in a project by generating and feeding inputs into the target program. Two prominent branches of greybox fuzzing are coverage-based greybox fuzzing and directed greybox fuzzing.

Coverage-based Greybox Fuzzing. Coverage-based fuzzing has emerged as a popular approach within the field of greybox fuzzing. Its main objective is to maximize code coverage and explore various execution paths to identify potential vulnerabilities. Coverage-based fuzzing has been enhanced with lots of techniques (e.g., static and dynamic program analysis [13, 18, 21], intelligent optimal strategies [10, 24, 36]), and has successfully discovered numerous vulnerabilities in real-world scenarios.

However, for real-world programs, the codebase is often large and complex, and a significant portion of the code remains unchanged over time. Fuzzing this large but unchanged code is less effective in finding vulnerabilities. Research by Zhu et al. [39] highlights that about four out of five reported bugs in OSSFuzz are introduced by recent code changes. While fuzzing the entire codebase is undoubtedly important, it is necessary to consider the limited availability of resources and the higher likelihood of newly added code introducing vulnerabilities. Therefore, prioritizing efficient fuzzing of the newly added code is crucial to effectively detect bugs.

Directed Greybox Fuzzing. Directed greybox fuzzing focuses on testing specific parts of a program, and there has been significant research dedicated to addressing challenges in target selection and improving the efficiency of reaching the target [14, 17, 30, 35].

For target selection, different applications require different settings. For patch testing, the target code can be manually specified as the patch site [17]. For crash reproduction, the target code can be set as the crash site [5]. For static analysis report verification, natural language processing techniques can be used to set the target code [35]. Additionally, the target code can be set as potential buggy points indicated by various sanitizers [27]. In the context of finding bugs introduced by commits, our work straightforwardly sets the commit change sites as the original targets.

Many existing approaches aim to improve the efficiency of reaching the target in directed greybox fuzzing. AFLGo [5], one of the pioneering works in this field, defines a distance metric to measure the proximity of the testing inputs to the target code. It calculates the distance between a testing input and a target code by averaging the distances between executed blocks and the target. This approach helps identify inputs that are closer to the target, and a power scheduling scheme is proposed to prioritize these inputs. Several subsequent works [7, 11, 17] have followed this line of research.

Taking account of the reachability analysis may also improve the fuzzing efficiency. By identifying inputs that cannot reach the target and, therefore, cannot trigger the correlated vulnerability, these inputs can be discarded. Some works employ deep learning methods to pretrain a model that predicts whether an input can reach the target [40]. Other works use static analysis to identify inputs that are path-wise or constraint-wise not reachable or relevant to the target, and subsequently discard them [14, 23].

DGF for Commit Testing. As highlighted by regression greybox fuzzing (RGF) [39], new commits are often prone to introducing bugs into a program. Therefore, it is crucial to efficiently detect whether a commit has introduced a bug or not. As a promising countermeasure, DGF can be leveraged to address this objective. However, there are two main problems in applying DGF to this task.

The first problem is that the crash sites of the introduced bugs may not be the same as the commit change sites. In our analysis, we found that over *half* of the commit-introduced bugs have change sites different from the crash sites, as depicted in Table 1. Existing approaches primarily focus on reaching the target, while not providing substantial assistance for thoroughly testing the target after it has been reached, thus resulting in miss detection of the potentially introduced bugs. The second problem arises from the fact that a commit often modifies multiple sites, resulting in multiple targets for

```
1  static int correct_orientation(struct i *image, ...){
2    //...
3    rotateImage(image, ..., True); // can toggle
4  }
5  static int processCropSelections(struct i *image, ...){
6    // only cropped image sections are rotated
7    rotateImage(image, ..., False); < can't toggle
8    if(img_mode == COMPOSITE_IMAGES){
9      //...
10   }else{
11     // read data from image
12     extractSeperateRegion(image, ...); < crash site
13   }//...
14 }
15 static int createCroppedImage(struct i *image,...){
16   rotateImage(image, ..., True); // can toggle
17   //...
18 }
19 static int rotateImage(struct i *image,..., int rotIP){
20   if(rotIP){
21       res_temp = image->width;
22       image->width = image->length;
23       image->length = res_temp;
24   }
25   //...
26 }
```

Figure 1: Motivating example.

DGF. However, existing approaches may overlook certain targets and compromise the individual directness required to thoroughly test each target.

2.2 Motivating Example

To better illustrate the aforementioned problems, we provide an excerpt, as shown in Figure 1, highlighting a specific issue (i.e., #519) related to the `rotateImage` function in the libtiff library [2]. In Figure 1, the bold texts are patch codes.

In the `rotateImage` function, there is a problem where the width and length parameters of the main image are mistakenly toggled, causing issues in subsequent executions (Figure 1, line 12). Specifically, when calling the `extractSeparateRegion` function, the main image's dimensions are incorrectly modified by the previous call of the `rotateImage` in line 7. Due to this modification, the program ends up reading data from the main image using the incorrect width and length parameters. As a consequence, a heap buffer overflow is triggered during the subsequent function call to `extractContigSamplesBytes` (not shown in Figure 1 for brevity).

If the fuzzer only focuses on reaching the toggle code in lines 21 - 23 of the `rotateImage` function, it will struggle to trigger the bug. This is because distance-based fuzzers will prioritize fuzzing other functions (`correct_orientation` in line 1 and `createCroppedImage` in line 15) that are closer in distance, generating more seeds that can reach the target but not trigger the bug. This loss of diversity can limit the effectiveness of bug detection.

Similarly, reachability-based fuzzers will fail to find the bug due to their design limitations in testing the affected code beyond the target site. Recent work, such as SelectFuzz [23], also struggles to trigger the bug because it only instruments target-related code before reaching the target, losing important feedback to generate or save inputs that explore the *else* branch at line 10.

The developer fixed this issue by adding a new parameter called *rotIP*, which serves to determine whether the image should be toggled or not. Now, when testing whether this patch commit introduces any bug, we face the challenge of dealing with multiple targets. In the given code, lines 3, 7, 16, and 19 - 24 all represent different change sites (targets) that need to be thoroughly tested for potential bugs.

To overcome these challenges, improved approaches are needed to prioritize input diversity, extend assistance beyond target reaching, and handle multi-target scenarios. By doing so, we can enhance the effectiveness and efficiency of directed fuzzing in identifying newly introduced bugs.

3 Design of WAFLGo

In this section, we present the design details of WAFLGo, a directed fuzzer specifically designed to efficiently identify bugs introduced by commits. First, we describe how to select fuzzing targets from the commit diff and calculate distance for each target. Second, we discuss how to define and identify the critical code. Finally, we discuss how to use the critical code to guide our tool during fuzzing. The overall architecture of WAFLGo is illustrated in Figure 2.

3.1 Target Selection and Distance Calculation

When testing a commit to discover potential vulnerabilities, it is necessary to test every change site and its affected code thoroughly. However, commits often consist of numerous change sites, resulting in multiple fuzzing targets. Handling each change site separately would be impractical and inefficient as it disregards some connections between them. For example, the similar preconditions shared by change sites within the same function. On the other hand, treating all change sites as a single target, as done in existing works [5,7,17], leads to the oversight of certain change sites and compromises individual directness.

In this work, we propose a novel multi-targets distance metric for commit testing. This distance metric merges the multiple change sites into a smaller set of function-level targets. The rationale behind this consolidation is to leverage shared preconditions among the same group of changes. Next, the distance metric calculates an individual distance for each merged target, ensuring the preservation of directness for each target, and pay more attention to the rarely executed targets.

1) Target Merging. We select the commit change sites as the initial targets. The functions that contain these initial targets are referred to as *target functions*. As the changes in one function share lots of similar preconditions, we merge all the change sites in the same function as one. We first

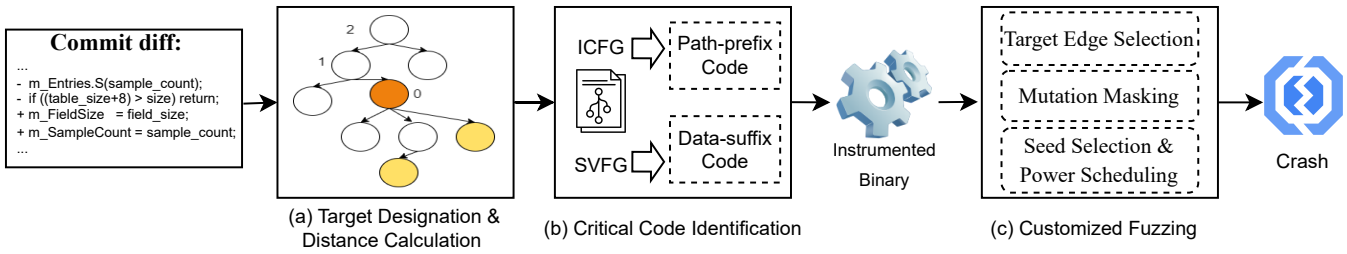


Figure 2: Architecture of WAFLGo.

construct the dominator trees of the target functions and then set the immediate dominator of each initial target in the target function as the merged target for that function. This ensures that each target function has only one target, simplifying the distance calculation phase and making it more efficient. The initial targets then become the descendants of the merged target, which will be marked and explored in the fuzzing phase.

2) Distance Computation. The basic blocks that contain merged targets are referred to as *target blocks* T_b. The *block distance* $d_b(m, T_b)$ determines the distance from a basic block m to each target block T_b. More formally, we define the distance $d_b(m_1, m_2)$ between two basic blocks m_1 and m_2 as the minimum number of basic blocks along the shortest path between them. As we have built the inter-procedural control flow graph (ICFG), we can more precisely compute the distance across functions than AFLGo [5]. Additionally, we assign an *edge distance* $d_e(e, T_b)$ to every edge e by calculating its distance to the target block T_b. Specifically, for an edge e_{ij}, where i and j respectively denote its head and tail nodes, we compute its distance as follows:

$$d_e(e, T_b) = d_b(j, T_b) \tag{1}$$

Taking Figure 3 as an example, we set line 14 as the target. In this case, we define $d_{e_{ab}}(e_{ab}, g) = d_b(b, g) = 2$. The edge distance is used for target edge selection in Sec 3.3.1.

Let $\delta(s)$ be the execution trace of a seed s. This trace contains the executed basic blocks of s. For each target T_b, we define the input distance $d_s(s, T_b)$ for a seed s as:

$$d_s(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|} \tag{2}$$

$$\xi(s) = \{m \mid m \in \delta(s) \text{ and } d_b(m, T_b) \neq \text{NaN}\} \tag{3}$$

The input distance of $s : a \to b \to d$ in Figure 3 is $\frac{3+2}{2} = 2.5$. In the multi-targets scenario, to ensure that each target receives sufficient testing attention, when using input distance for power scheduling in Sec 3.3.2, we focus on the rarest executed target T_r by using $d_s(s, T_r)$ as the input distance.

3.2 Critical Code Identification

WAFLGo highlights that in the context of commit testing, the successful identification of new vulnerabilities requires more than simply reaching the commit change site (target). It is crucial to conduct thorough testing of the affected code by employing diverse inputs. This is because the target can be accessed through various program states, including different paths and data values. Furthermore, the impact of the target may extend throughout the entire program, potentially leading to vulnerabilities that manifest at distant locations.

To ensure thorough testing of the commit change site, it is crucial to extract additional information both before and after the target. This information serves two key purposes: guiding the fuzzer toward the target change site and providing guidance for generating high-quality inputs that explore different program states and the affected code. Specifically, we identify code that is control- or data-dependent on the initial change sites as critical code. This critical code is further classified into two categories: *path-prefix* code and *data-suffix* code.

1) Path-prefix code. The path-prefix code refers to all the code that can lead to the execution of the target code. It includes the code blocks that are directly or indirectly connected to the target code in the ICFG. Consequently, the code that acts as a data-prefix (i.e., data-dependency predecessor) is inherently included within the path-prefix code. For instance, in Figure 3, the path-prefix code for target g consists of code blocks a, b, e, and f.

2) Date-suffix code. The data-suffix code includes the code that depends on the critical variables used in the initial targets. To identify the data-suffix code, we employ the following strategy in conjunction with data-flow analysis.

We exclusively consider variables used in the target that are **written** as sources for taint analysis. In the context of Figure 3, which features two variables, namely x and y, our focus for tracking the data-suffix code is limited to variable x due to its being written. For the inter-procedure taint analysis, when a tainted variable is employed as formal parameters, we taint both the callee's actual arguments and the callee's return values. Similarly, if a tainted variable is utilized as a return value, we taint the corresponding variable at the caller's call site.

Subsequently, we collect the corresponding basic blocks that use the tainted variables, constituting the data-suffix code. For instance, in Figure 3, the data-suffix code comprises code blocks i and k.

It's noteworthy that the data-suffix code yields more comprehensive and meaningful insights compared to the path-

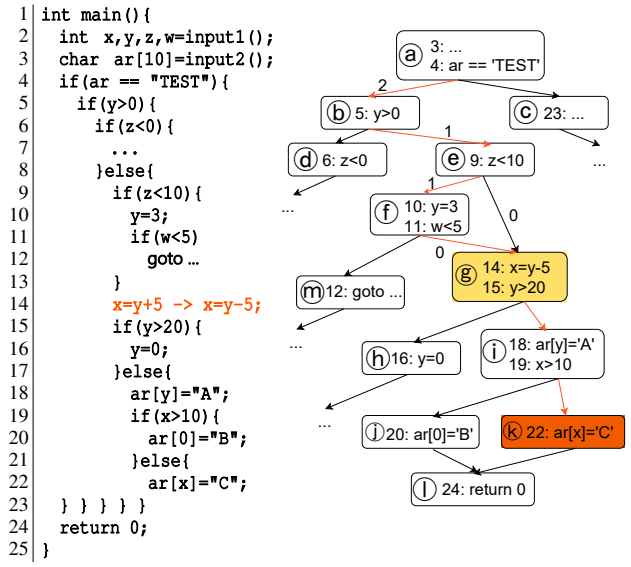


Figure 3: Illustration example. Line 14 is the change site (target) where we change $x = y+5$ to $x = y-5$. Line 22 is the crash site.

suffix code which often used for code coverage measurement. This distinction arises because not all the path-suffix code reached by the target is affected by it (e.g., blocks h, j and l in Figure 3). Conversely, the data-suffix code directly reflects the impact of the target on the program by focusing on the specific data changes by the target.

3.3 Critical Code Guided Fuzzing

Algorithm 1 sketches out our fuzzing procedure. In this subsection, we will demonstrate our target edge selection mechanism (line 7), and the practical application of the modified mutation operations through mutation masking (line 9). Then, we present our adaptive seed selection and power scheduling mechanisms (line 4 and line 8).

3.3.1 Input Generation Strategy

In fuzzing, the generation of inputs that can trigger crashes is highly prioritized. However, in the context of commit testing, two additional criteria are crucial for generating high-quality inputs. First, it is essential to generate inputs that can efficiently reach the target. Second, once the target is reached, we have to maintain the reachability, and then generate diverse inputs to explore different program states and the affected code.

To meet the criteria, WAFLGo utilizes a novel critical code guided input generation strategy that involves target edge selection and mutation masking. Specifically, WAFLGo selects a target edge for each input based on its trace and generates a mask for the input to maintain the execution of the target edge. Our insight is that when we have a seed that

Algorithm 1 WAFLGo Algorithm

Input: *Prog*, *Seeds*
Output: *potential bugs*

```
1:  function FuzzOne(Prog, Seeds)
2:      Queue ← Seeds
3:      while true do
4:          input = SelectOne(Queue)
5:          test_target = GetRarestTarget(Prog, input)
6:          test_edge =
7:                  GetTargetedge(Prog, input, test_target)
8:          score = AssignEnergy(input)
9:          mask = ComputeMask(Prog, input, test_edge)
10:         for i = 1 to score do
11:             newinput = Mutate(input, mask)
12:             Execute(Prog, newinput)
13:         end for
14:     end while
15: end function
```

exercises critical code, we aim to generate diverse inputs from this seed while ensuring the execution of the critical code. This means that our fuzzer should be aware of the important preconditions established by the seed and avoid unintentional side-effects during seed mutation that could deviate from our target.

1) Target Edge Selection. Based on the execution of the input being fuzzed, the target edge selection process varies. If the input does not reach the target, WAFLGo enters the target reaching stage. If the input reaches the target, WAFLGo enters the comprehensive testing stage. Algorithm 2 outlines how WAFLGo select target edge.

Target reaching stage. We leverage the path-prefix code identified in Sec 3.2 to systematically guide the fuzzer toward the target in a gradual and incremental manner. To achieve this, we take a step forward by preserving all incoming edges of the path-prefix code, denoted as "Prefixes" in Algorithm 2. This decision is motivated by the fact that edge execution state provides fine-grained information compared to basic blocks. For example, when inputs execute either edge e_{eg} or e_{fg}—both leading to the execution of block g—they correspond to distinct program states. The edges within the "Prefixes" list are organized in ascending order based on their edge distance. In the Figure 3, the "Prefixes" is $(e_{fg}, e_{eg}, e_{ef}, e_{be}, e_{ab})$.

If an input has not triggered any of the targets (Algorithm 2, lines 2 - 12), our approach aims to guide the fuzzer first to reach the rarest target. To achieve this, we select the *closest* edge that has been *triggered* by the input as the target edge. We systematically examine each edge in the "Prefixes" list. To avoid repeatedly generating inputs to execute the same edge, if an edge has been selected more times than a specified threshold (line 3), we consider it fully tested and skip to the next edge. We determine if the input has triggered each edge by evaluating its trace bitmap, which is a shared memory holds by fuzzer to record all the edges an input exercised. Once

Algorithm 2 Target edge Selection in WAFLGO

Input: *Prog*, *input*, *test_target*,
Prefixes, // prefix edges of each target in ICFG, sorted by distance
Suffixes // suffix edges of each target in data-flow
Output: *target_edge for input*

```
1:  function GetTargetEdge(Prog, input, test_target)
2:      for edge ∈ test_target.Prefixes do
3:          if selectcount[edge] < threshold then
4:              if edge ∈ input.trace then
5:                  target_edge = edge
6:                  selectcount[edge]++
7:                  break
8:              end if
9:          end if
10:     end for
11:     if input not trigger target and ! target_edge then
12:         target_edge = -1
13:     else if input triggers target and ! target_edge then
14:         for edge ∈ test_target.Suffixes do
15:             if edge ∈ input.trace then
16:                 target_edge = min(hitcount[edge])
17:             end if
18:         end for
19:     end if
20:     return target_edge
21: end function
```

we find a triggered edge, we designate it as the "triggered-closest" edge, which becomes the target edge for further mask generation. If none of the edges in the "Prefixes" list have been executed and the target has not been triggered, we assign the value of -1 to the *target_edge* variable. This indicates that the seed is considered highly irrelevant to the target. As a result, no mask is generated for this input, and only havoc mutation is performed.

Take Figure 3 as an example, if we have a seed A with the input trace: $a \to b \to e \to f \to m$, we first select e_{ef} as the target edge. If the $selectcount[e_{ef}]$ exceeds $threshold$, we will select e_{be} as the target edge and so on.

Comprehensive testing stage. In this stage, we leverage both the path-prefix code and the data-suffix code to enhance input diversity. Simultaneously, we utilize the data-suffix code to direct the fuzzer towards the affected code. Similar with "Prefixes", we define "Suffixes" as a set of all incoming edges of the data-suffix code. In Figure 3, the "Suffixes" is $\{e_{gi}, e_{ik}\}$.

If an input has triggered one target, we still prioritize selecting the "triggered-closest" edge, as we aim to generate diverse inputs that can reach the target. Then, if all the executed edges of the input in the "Prefixes" list have been triggered with sufficient frequency, we proceed to steer the fuzzer toward low-frequency code that is affected by the target. Specifically, we identify the rarest hit edge among the previously discovered "Suffixes" that the input executed (lines 13 - 18). This rarest hit edge is then designated as the target edge for subsequent mask generation.

Consider Figure 3 as an example. Suppose we have a seed B with the input data: $x = 0, y = 16, z = 11, w = 0$, and $ar =$'TEST', leading to the input trace: $a \to b \to e \to g \to i \to j \to l$. Initially, we select e_{eg} as the target edge, followed by e_{be} and e_{ab}. Subsequently, we choose the rare hit edge e_{ik} in the "Suffixes" as the new target edge.

2) Mutation Mask. To maintain the execution of the target edge, we incorporate a mutation mask mechanism inspired by techniques employed in FairFuzz [18] and GreyOne [13]. Specifically, WAFLGO employs a byte-by-byte mutation process on the input using a predefined set of mutation operators, which includes bit-flip, insertion, and deletion. After each mutation, WAFLGO checks whether the target edge is still executed. If the target edge is still hit, WAFLGO identifies the corresponding byte in the input as mutable and updates the input's mask accordingly. In subsequent mutations, such as the havoc mutation, the mutable bytes of the input will be mutated with higher probability to generate new inputs.

3.3.2 Seed Selection and Power Scheduling

The traditional strategy of appending new inputs to the end of the seed queue and selecting them sequentially can prove inefficient, particularly when a significant portion of the inputs in the queue are unrelated to the target, as discussed in [23]. To overcome this inefficiency, expedite target reachability, and ensure thorough testing of the affected code, we introduce a prioritization approach. This approach emphasizes seeds that trigger new critical code during seed selection. Specifically, we maintain a favored queue with higher priority and add seeds that meet this criterion to it. During the fuzzing phase, WAFLGO has a greater chance of selecting a seed from the favored queue.

Regarding power scheduling, relying solely on the distance metric of "Prefixes" as traditional works may reduce the diversity of the generated inputs, as it consistently prioritizes closer inputs. WAFLGO assigns energy to input differently based on whether the input reaches the target. For inputs that do not reach the target, WAFLGO adopts the simulated annealing algorithm proposed in AFLGo, with a modification in the energy calculation using the new seed distance metric. Specifically, it gradually assigns more energy to inputs that are closer to the rarest executed target, facilitating faster target reaching. On the other hand, for inputs that reach the target, WAFLGO adjusts its approach by disregarding the distance metric for energy assignment. Instead, it focuses on the code coverage of the data-suffix code. This means that more energy is assigned to inputs that cover more data-suffix code, resulting in the generation of more inputs for thorough testing of the affected code.

4 Implementation

The implementation of WAFLGo mainly consists of two parts: the static analysis module and the dynamic fuzzing module. For the static analysis module, we utilize the SVF static analysis framework [31] to construct an ICFG and an inter-procedural static value-flow graph (SVFG) from LLVM IR. The static analysis module is implemented with about 1,200 lines of C/C++ code. The dynamic fuzzing module is implemented based on AFLGo [5], with about 2,000 lines of C code. Next, we discuss a few important implementation details.

Initial Target Acquisition. When a commit is submitted, WAFLGo utilizes the `git diff` command to retrieve the differences between commits. For added lines, the change site is directly stored as "filename:linenum", indicating the precise location of the added code. For deleted lines, WAFLGo selects nearby code lines for further analysis. Next, to ensure effective testing of the commit, WAFLGo filters out change sites that are not relevant to the program's functionalities, such as comments or changes pertaining to other programs within the project. To achieve this, a simple script is employed to perform a straightforward reachability analysis, determining if the commit changes can be reached from the entry point of the tested program. In the end, only the change sites that truly affect the tested program are tested.

Inter-procedural Data-flow Analysis. In the critical code identification phase, WAFLGo utilizes a forward inter-procedure data-flow analysis to identify the data-suffix code. The process involves identifying the variables used in the change sites and then traversing the SVFG to collect the code that exhibits data dependency on these variables. However, due to the potential presence of multiple callers for a callee, it is essential to accurately propagate the data dependency to the appropriate call site. In this regard, WAFLGo implements a *context-sensitive* propagation strategy that selectively propagates the data dependency to the call site responsible for transferring the data-dependent parameters to the callee. This ensures that only the relevant call site is marked as data-dependent, enhancing the accuracy of the critical code identification process.

Extra Feedback Utilization. Existing fuzzers often discard inputs that do not execute any new branches, assuming that they are not valuable for further exploration. However, in the context of directed fuzzing, the branches executed by non-target reaching inputs can still be valuable if they are executed by target reaching inputs. WAFLGo acknowledges the importance of the new feedback provided by target reaching inputs. Therefore, WAFLGo constructs an additional trace bitmap to record the execution traces of the inputs that successfully reach the targets. This allows WAFLGo to leverage the information from target reaching inputs and further guide the exploration process for more effective bug detection.

Table 1: Real-world benchmark programs. "DiffLoc" and "SameLoc" are used to denote whether the change sites of the BIC are distinct from or identical to the crash sites, respectively. "BIC Hash" represents the bug-inducing commit hash.

Project	Program	Diff Loc	Same Loc	BIC Hash	Total	Path-prefix	Data-suffix
		21	9		BBNum	BBNum	BBNum
Libtiff	tiffcrop	#488		7057734d	13921	7781	1253
	tiffcrop	#498		07d79fcac	15256	8166	1866
	tiffcrop	#519		f13cf46b	15227	7389	5
	tiffcrop		#520	e3195080	15706	12039	4992
	tiffcrop	#527		07d79fcac	15256	8166	1687
	tiffcrop	#530		f13cf46b	15227	7389	5
	tiffcp		#548	3079627e	13134	9995	6251
	tiffinfo		#559	b90b20d3	13722	7729	1493
Bento4	mp4info	#652		c9f2c53	15621	6002	417
	mp4info		#679	2e29350	17216	1262	7600
	mp4audioclip	#732		bbb6f24	14593	5885	1347
	mp42aac		#751	61b2012	14424	4949	149
Mujs	mujs	#65		8c27b126	6482	52	7
	mujs	#141		832e0690	6996	4240	631
	mujs		#145	4c7f6be	7319	939	6833
	mujs	#166		3f71a1c9	15791	10558	1756
Libjpeg	cjpeg	#493		88ae609	4982	468	106
	jpegtran	#636		88ae609	6075	898	429
Tcpreplay	tcprewrite		#702	0a65668a	4110	2925	1606
	tcprewrite	#718		2c76868d	4030	3005	1742
	tcpprep	#756		16442ac3	1855	921	687
	tcpreplay	#772		4f9158da	2240	1234	7
Libxml2	xmllint	#535		9a82b94a	66472	13030	5202
	xmllint	#550		7e3f469b	66150	9075	147
Poppler	pdfunite	#1282		3d35d209	44103	6358	82
	pdfunite		#1289	3cae7773	1015	497	12
	pdftops	#1303		e674ca64	42235	7700	373
	pdftoppm	#1305		aaf2e808	37682	5959	77
	pdftoppm		#1381	245abada	51098	6557	122
ImageMagick	magick	#6075		a107b941	134594	9871	1206

5 Experimental Setup

5.1 Research Questions

RQ1 How effective is WAFLGo in discovering bugs introduced by commits?

RQ2 Does the guidance toward critical code improve the efficiency of fuzzing?

RQ3 Does the multi-target optimizations improve the efficiency of fuzzing?

RQ4 Can WAFLGo detect new vulnerabilities in real-world programs?

5.2 Commit Dataset

5.2.1 Commit Dataset for Testing Known Bugs

We construct a commit dataset to answer the RQ1, 2, and 3.

Real-world Target Programs. We conduct our evaluation in real-world programs that have been frequently evaluated in the existing fuzzing frameworks [5, 11, 13, 14], which are shown in Table 1. The selected programs exhibit diverse functionalities and vary in size, allowing us to demonstrate the effectiveness and scalability of our approach in different contexts.

Bug Selection Criteria. To ensure the reliability and relevance of our commit benchmark subjects from real-world

Table 2: Compared fuzzers.

Fuzzer	Category	Description
AFLGo	Direct	Sophisticated seeds prioritization
Windranger	Direct	Deviation basic blocks
SelectFuzz	Direct	Selective path exploration
FishFuzz	Direct	Multi-targets fuzzing
AFL	Coverage	Evolutionary mutation strategies
AFL++	Coverage	Optimization of overall fuzzing framework
FairFuzz	Coverage	Mask mutation strategy
AFLChurn	Regression	Code history based strategy

programs, we establish the following criteria to collect the most recent closed and reproducible issues (as of Sep'10).

- Fixed Issue. The selected issue must be marked as "fixed", with the bug-fixing commit identified and linked.This aids in identifying the BIC of the bug.
- Executable Proof-of-Concept (PoC). The issue should have an accompanying executable PoC, enabling the verification and reproducibility of the bug. This ensures that the bug can be tested and verified in a practical manner.

DGF Target Code Designation. For each issue listed in Table 1, we manually identify its BIC, and designate the targets as the change sites of its BIC. The process begins by manually finding a good commit (*GoodC*) in the commit history where the bug is absent. Then we use `git bisect` to systematically locate the first crash commit (*FirstC*) triggered by the PoC. Following this, we perform a manual verification to confirm whether *FirstC* indeed serves as the BIC. This verification entails an examination of the bug-fixing commit and its relationship with *FirstC*. We also use debugging tools like gdb to understand how changes in *FirstC* impact the execution of the PoC. In instances where `git bisect` fails to accurately identify *FirstC*, which occurs due to configuration problems in some cases, we resort to an alternative method. We manually trace the blame information for the change sites of the bug-fixing commit to determine the BICs for these particular cases.

5.2.2 Commit Dataset for Finding New Bugs

To answer the RQ4, we apply WAFLGo to detect new vulnerabilities.

We include the projects evaluated in recent fuzzing works as our testing objects and compile their programs using default configurations. We test the latest 50 commits of these projects and designate specific commit change sites as the target code. To accommodate practical constraints, we allocated a maximum of 24 hours for testing each commit.

5.3 Compared Tools

We compare WAFLGo with the fuzzers listed in Table 2.

Directed Fuzzing Tools. We implement commit-directed fuzzing into AFLGo. In order to evaluate the effectiveness of our approach, we select AFLGo as our baseline for comparison. Besides, we add the state-of-the-art DGFs for comparison, including Windranger [11], SelectFuzz [23] and FishFuzz [38] which stand for distance-based, reachability-based and multi-target DGFs, respectively.

Coverage-based Fuzzing Tools. In order to compare commit-directed fuzzing to traditional coverage-based fuzzing, we choose the base AFL [1], state-of-the-art tool AFL++ [12] and innovative tool Fairfuzz [18] for comparison. AFL and AFL++ demonstrate the effectiveness of coverage-based fuzzers in detecting newly introduced bugs through commits. FairFuzz utilizes a novel mutation mask that biases mutations toward producing inputs that hit a given rare branch, which is similar to the mask design we used in our approach.

Regression Fuzzing Tools. We add AFLChurn [39] to our comparison because it leverages the observation that recently modified code in commits should receive more attention from the fuzzer. AFLChurn introduces a new power scheduling mechanism to effectively test the modified code.

5.4 Configuration

In this subsection, we present some necessary configurations.

Initial Seeds. The initial seeds can greatly affect the efficiency of the fuzzing process. To ensure a fair comparison, we apply the same seeds per bug across different fuzzers. In our experiment, we utilize the seeds provided by the project itself and those available in UNIFUZZ [20] for the corresponding input formats.

Time Budget. We limit each experiment to a time budget of 24 hours and repeat it five times. This decision is justified by the need for rapid iteration and security left-shifting in the development process. The goal is to conduct thorough testing of the code before it is merged, prioritizing early detection of potential issues rather than relying solely on post-merge testing. Considering the practical constraints and the need for efficient bug detection, testing a commit for more than 24 hours would be impractical.

Infrastructure. All experiments are conducted in a docker container configured with 1 CPU core of 2.40GHz E5-2680 V4 and the 64-bit Ubuntu 16.04 LTS. In total, we spend several weeks running evaluations on four servers, each of which has an Intel Xeon E5-2650 v4 (2.2GHZ, 48cores) CPU, and 256 GB memory.

6 Performance Evaluation

6.1 Efficiency of WAFLGo

We measure the time used by WAFLGo to reproduce the issues. We present the evaluation results in the 4th column of Table 3. In general, WAFLGo effectively reproduces 21 issues out of the 30 cases, achieving the highest success rate

Table 3: The average crash exposure time of WAFLGo and the compared fuzzers. Time-to-Exposure indicates the reproduction time (hour) averaged over five runs. T.O. indicates the fuzzer cannot reproduce the issue within the given time budget, 24 hours. For the timeout cases, we take the Time-to-Exposure time as 24 hours to calculate the speedup.

No.	Issue-id	Program	Time-to-Exposure(hour)									Factor							
			WAFLGo	AFLGo	Wind.	Selc.	Fish.	AFL	AFL++	Fair.	AFLC.	AFLGo	Wind.	Selc.	Fish.	AFL	AFL++	Fair.	AFLC.
1	#488	tiffcrop	6.247	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	7.956	T.O.	3.8	3.8	3.8	3.8	3.8	3.8	1.3	3.8
2	#498	tiffcrop	0.001	0.011	0.004	0.003	0.012	0.005	0.005	0.001	0.003	9.5	3.9	2.6	10.4	4.6	4.4	1.0	2.5
3	#519	tiffcrop	0.286	6.059	3.002	2.081	3.955	6.426	0.613	0.617	10.958	21.2	10.5	7.3	13.8	22.4	2.1	2.2	38.3
4	#520	tiffcrop	0.940	3.230	1.301	1.230	T.O.	6.080	2.305	5.367	1.913	3.4	1.4	1.3	25.5	6.5	2.5	5.7	2.0
5	#527	tiffcrop	13.903	T.O.	T.O.	17.596	17.354	T.O.	16.071	19.717	T.O.	1.7	1.4	1.3	1.2	1.7	1.2	1.4	1.7
6	#530	tiffcrop	9.759	T.O.	19.842	T.O.	15.428	T.O.	T.O.	T.O.	13.340	2.5	2.5	2.5	1.6	2.5	2.5	2.5	1.4
7	#548	tiffcp	2.593	23.401	14.723	11.489	T.O.	22.143	3.610	9.008	T.O.	9.0	5.7	4.4	9.3	8.5	1.4	3.5	9.3
8	#559	tiffinfo	0.656	1.826	4.906	14.600	T.O.	2.726	2.617	1.084	5.509	2.8	7.5	22.3	36.6	4.2	4.0	1.7	8.4
9	#732	mp3aud.	0.010	0.134	0.069	0.050	0.064	0.055	0.076	0.062	0.055	13.0	6.7	4.9	6.3	5.4	7.4	6.0	5.4
10	#751	mp42aac	12.617	T.O.	T.O.	T.O.	T.O.	T.O.	14.768	T.O.	T.O.	1.9	1.9	1.9	1.9	1.9	1.2	1.9	1.9
11	#145	mujs	0.019	0.082	0.069	0.669	6.789	0.100	0.087	0.100	0.104	4.4	3.7	35.5	359.9	5.3	4.6	5.3	5.5
12	#493	cjpeg	0.028	0.509	0.633	0.158	0.523	0.831	0.850	3.900	0.368	17.9	22.2	5.6	18.4	29.2	29.9	137.0	12.9
13	#636	jpegtran	0.016	0.054	0.100	0.043	0.082	0.021	0.019	0.050	0.051	3.5	6.4	2.7	5.2	1.4	1.2	3.2	3.2
14	#702	tcprewrite	0.124	1.012	1.955	0.150	0.236	1.160	1.557	1.265	0.679	8.2	15.8	1.2	1.9	9.3	12.5	10.2	5.5
15	#718	tcprewrite	0.714	1.514	1.651	0.285	1.063	8.977	3.119	3.257	8.539	2.1	2.3	0.4	1.5	12.6	4.4	4.6	12.0
16	#756	tcpprep	0.401	6.671	0.535	1.746	0.867	6.881	T.O.	3.097	6.722	16.7	1.3	4.4	2.2	17.2	59.9	7.7	16.8
17	#772	tcpreplay	0.027	0.076	0.173	0.157	0.071	0.071	0.026	0.070	0.070	2.8	6.4	5.8	2.6	2.6	0.9	2.6	2.6
18	#535	xmllint	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
19	#1289	pdfunite	0.382	T.O.	1.891	1.716	13.155	T.O.	0.651	12.811	11.441	62.8	5.0	4.5	34.4	62.8	1.7	33.5	30.0
20	#1305	pdftoppm	6.672	11.891	17.470	T.O.	16.537	12.901	14.154	12.382	11.218	1.8	2.6	3.6	2.5	1.9	2.1	1.9	1.7
21	#6075	magick	10.989	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
#Reproduced / Average			21	15	17	16	15	15	17	18	16	9.1	5.4	5.7	25.8	9.9	7.2	11.2	8.0

among all the compared fuzzers. As the second most successful fuzzer, FairFuzz, reproduces 18 issues out of the 30 cases. For all the reproduced issues, WAFLGo outperforms other fuzzers, achieving an average speedup of 10.3×. WAFLGo fails to reproduce nine issues (i.e., Bento4 issue #679 and #652, Mujs issue #166, #141 and #65, Poppler issue #1282, #1305 and #1381 and Libxml2 issue #550), within the time budget of 24 hours, while other fuzzers could not trigger them, neither. The main reason is that these bugs require satisfying complex path constraints, which is beyond the scope of our approach. We further analyze these issues and find that none of the test cases generated by any of the fuzzers are able to reach the change site, let alone trigger the crashes.

Comparison with Directed Fuzzers. Compared with the baseline tool AFLGo, which WAFLGo is built upon, WAFLGo identifies six more bugs, with an average speedup of 9.1×. As for other state-of-the-art directed greybox fuzzers, namely Windranger, SelectFuzz, and FishFuzz, WAFLGo identifies four, five, and six more bugs, with an average speedup of 5.4×, 5.7×, and 25.8×, respectively.

Comparison with Coverage-based Fuzzers. We compare WAFLGo with three coverage-based tools listed in Table 2, namely AFL, AFL++, and FairFuzz. In total, WAFLGo identifies six, four, and three more bugs compared to them, with an average speedup of 9.9×, 7.2× and 11.2×, respectively. Our results show that our tool is more effective at detecting commit-inducing bugs compared to existing coverage-based fuzzers. This can be attributed to WAFLGo's capability of drawing more energy to test the code regions affected by commit changes.

Comparison with Regression Fuzzers. The evaluation results of AFLChurn are listed in the 12th column in Table 3. WAFLGo outperforms AFLChurn by successfully reproducing five more bugs. On average, WAFLGo achieves a speedup of 8.0×. The improved performance of WAFLGo is the result of its capability to identify fine-grained changes in the code compared to AFLChurn.

Directed Fuzzers v.s. Coverage-based Fuzzers. We notice that, in certain cases, directed fuzzers exhibit inferior performance compared to their coverage-based counterparts. There are two main reasons. First, in some scenarios, certain bugs are relatively easy to trigger, as demonstrated by cases such as issues #498, #732, and #535. These bugs can be successfully exploited within a short time frame of approximately one minute of fuzzing, as indicated in Table 3. Upon analysis, we find that the main reason for this is that the initial seed used for fuzzing is already in close proximity to the target location. This proximity allows for the generation of inputs that can effectively trigger the bug through simple mutations. In such cases, directed fuzzing-based tools may appear less effective, as the bug-triggering process does not require additional guidance or sophisticated mutation strategies. Second, some bugs, as exemplified by issue #519, have easily accessible target locations. This means that the majority of generated inputs can readily reach these targets. When the target is readily reached, the challenge shifts from reaching the targets to conducting thorough code testing, aligning more with a coverage-based testing task. Therefor, coverage-based fuzzers, such as AFL++

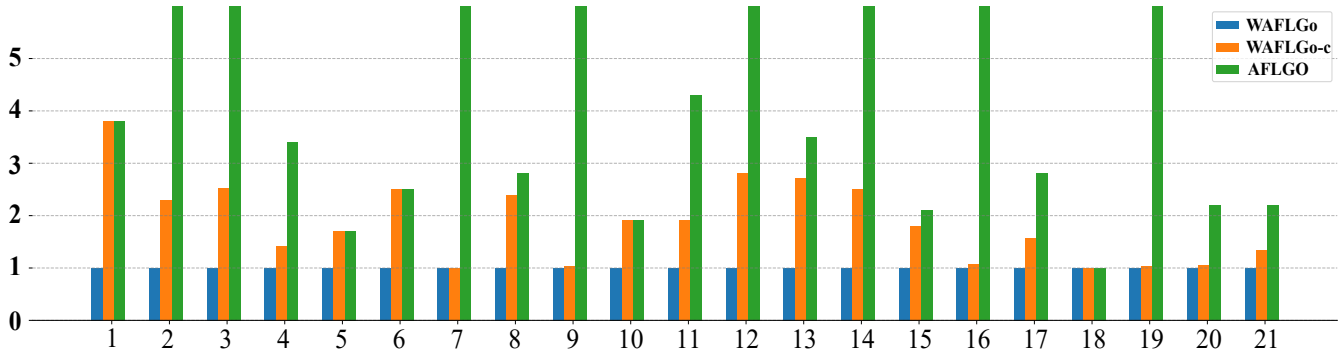


Figure 4: The component-wise effectiveness of WAFLGo. The y-axis represents the ratio of reproduction time, with WAFLGo as the baseline.

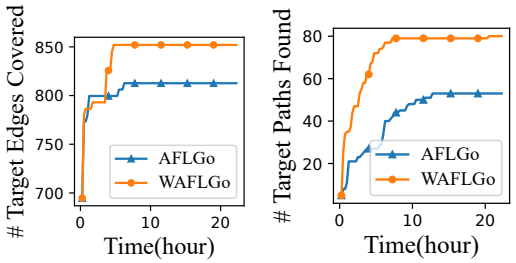
and FairFuzz, tend to excel in such scenarios. For example, in the case of issue #519, it is worth noting that more than 90% of seeds generated by all the fuzzers can successfully reach the target.

6.2 Impacts of Critical Code Guidance

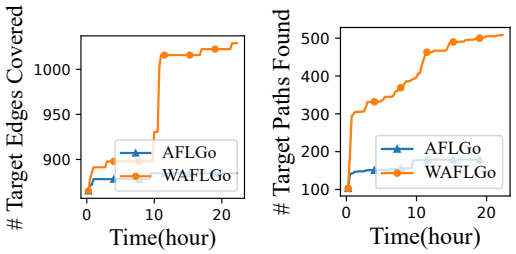
To effectively direct WAFLGo toward and thoroughly test the target and its affected code, we introduce a novel critical code guided input generation strategy. The core design includes a critical code identification method, a target edge selection and a mutation masking mechanism. To evaluate how the strategy contributes to the efficiency of WAFLGo, we created a variant of WAFLGo in which these mechanisms were disabled. We refer to this variant as WAFLGo-*c*, and we conducted a set of experiments to compare its performance with that of the original WAFLGo.

Bug Triggering Time. The experimental results are depicted in Figure 4. Overall, WAFLGo outperforms WAFLGo-*c* to reproduce the bugs with an average speedup of 2.1×. These results demonstrate the effectiveness of the critical code guidance. Notably, our strategy has shown remarkable effectiveness in certain cases, such as case No.1. Both WAFLGo-*c* and AFLGo fail to reproduce this bug. To further evaluate its performance, we extend the fuzzing time for AFLGo on case No.1 to 48 hours. As a result, AFLGo successfully detects the bug at 37.1 hours. However, we observe that even though AFLGo reaches the target within the first hour, the coverage of affected basic blocks in the subsequent fuzzing period increases very slowly. This indicates that after reaching the target, AFLGo does not pay much attention to testing the affected code, leading to the bug not being triggered within the given time budget.

Coverage of Affected Code. To further demonstrate the effectiveness of our target testing, we present the edge coverage achieved by AFLGo and WAFLGo, along with the count of paths that reach the targets in Figure 5. We exclude the coverage of inputs that do not reach the targets from our analysis, as they are unlikely to discover bugs introduced by the targets. Due to space limitations, Figure 5 showcases the results for



(a) Tcpreplay #718



(b) Poppler #1289

Figure 5: The edge coverage and the number of target-reached paths.

two issues: No. 15 (Tcpreplay #718) and No. 19 (Poppler #1289). We will make all of our data available on our website for further analysis and reference. From the results, we observe that WAFLGo achieves an average increase of 11.7% in edge coverage after 24 hours compared to AFLGo. Furthermore, WAFLGo discovers nearly 2× (181.5%) more paths than AFLGo after 24 hours. It is worth noting that during the initial three hours, WAFLGo discovered a greater number of paths reaching the target compared to AFLGo, which aligns with the findings shown in Figure 4, where WAFLGo identifies the bugs at a faster rate than AFLGo.

6.3 Impacts of Multi-Target Optimization

To address the multi-target challenge, we have proposed several optimizations, which include the distance calculation methods, the adjusted input generation strategy and the seed scheduling mechanism.

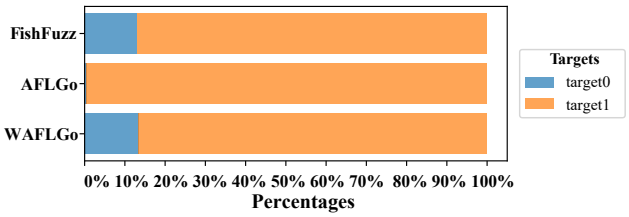


Figure 6: Target reached seeds.

Table 4: New vulnerabilities detected by WAFLGo

Program	Bug Type	Status	ID
tiffcrop	segmentation fault	patched	CVE-2023-3618
fig2dev	null pointer dereference	patched	CVE-2023-34629
fig2dev	segmentation fault	patched	CVE-2023-34630
fig2dev	memory leak	patched	CVE-2023-34631
swftophp	heap buffer overflow	reported	issue-271
swftophp	heap buffer overflow	reported	issue-270
swftophp	heap buffer overflow	reported	issue-269

Bug Triggering Time. In evaluating the effectiveness of our proposed optimization methods, we conducted a series of experiments involving WAFLGo-*c* and AFLGo. The experimental results shown in Figure 4 demonstrate that WAFLGo-*c* is faster than AFLGo for discovering bugs, achieving an average speedup of 4.3×. This improvement is attributed to the generation of high-quality seeds for each target using our multi-target optimizations. All 30 issues in our dataset have multiple targets, and 21 of them have more than 10 targets. Our target merging process reduces the number of targets by a factor of 6.72, and after the merging process, 20 of the issues have fewer than five targets.

Quality of Generated Seeds. To further understand the effectiveness of our multi-target optimization, we conducted an analysis of the seeds generated by AFLGo and WAFLGo for issue #1289. In this case, the BIC of issue #1289 modified one file with 31 additions and two deletions, resulting in changes to 32 basic blocks. Following our target merging process, WAFLGo identifies two targets. From Figure 6, we observe that nearly all the seeds generated by AFLGo reach target1 but ignore target0. However, it is target0 that represents the key change introducing the bug. Therefore, AFLGo fails to reproduce this bug. Furthermore, we analyze the seeds generated by FishFuzz, a fuzzer designed for multi-target scenario. Interestingly, we found that the seed distribution in FishFuzz is similar to that of WAFLGo, further highlighting the effectiveness of our approach in handling multi-target situations.

6.4 Detecting New Vulnerabilities

In this subsection, we employ WAFLGo to detect new vulnerabilities by applying them to various programs. WAFLGo successfully identifies seven new vulnerabilities during the time budget as listed in Table 4. It is noteworthy that WAFLGo demonstrated its effectiveness in uncovering vulnerabilities even in complex software such as *libtiff*. We promptly reported all identified vulnerabilities to the respective developers. As of the time of writing, four vulnerabilities have been patched and receive CVE IDs, showcasing the impact and value of our approach in contributing to the improvement of software security.

We further analyse the newly discovered vulnerabilities and found that the CVE-2023-34631 is introduced by the fixing commit (6678ad8) for CVE-2023-34630. The vulnerability originates from the code at `fig2dev/gengbx.c:1135` [3], where a linked list pointer is dereferenced without checking the existence of the next pointer. This led to a null pointer dereference bug (CVE-2023-34630). To address the vulnerability, the *sanitize_lineobject()* function in `fig2dev/read.c` was modified to include a check for the presence of the next pointer in the linked list. However, this adjustment affected the length of the linked list and disrupted other code logic, resulting in CVE-2023-34631 with invalid memory access. In order to resolve the vulnerability, the developers opted to revert the fix commit (6678ad8) and perform a careful refactoring of the linked list length handling in the *sanitize_lineobject()* function. It is worth noting that the change site of the bug-introducing commit is far away from the crash site, and it was not even evident in the stack trace of the crash. This observation further validates our hypothesis that effective fuzzing of newly committed code requires a focus not only on the changed code itself but also on the code that is affected by the changed code. By considering the broader code context, we can uncover potential vulnerabilities and bugs that may have cascading effects beyond the immediate scope of the modifications.

7 Discussion and Limitation

In this section, we discuss the limitations of WAFLGo and the possible future works.

Reaching Target Failure. WAFLGo proposes a comprehensive approach that focuses on thoroughly testing commit modifications and their affected code throughout the entire program. The first step in achieving this goal is to reach the modification sites. However, the task of reaching the target locations itself remains a significant challenge in the current research landscape of directed greybox fuzzing. As discussed in Section 6.1, there are instances where WAFLGo, along with existing fuzzers, fails to reach the modified sites and trigger the associated bugs. This limitation primarily stems from WAFLGo's difficulty in efficiently generating high-quality inputs that satisfy complex path constraints.

To address this challenge, incorporating advanced static analysis techniques could be highly beneficial. We plan to integrate other advanced static analyses (e.g., concolic symbolic

execution similar to [8, 16, 37]) into WAFLGo to mitigate this limitation.

Semantic Affected Code Identification. In WAFLGo, we collect all the code that exhibits a data dependency with the written variables used in the targets (commit changes) and consider it as the data-suffix code. However, not all variables in the commit changes hold the same level of importance regarding the potential introduction of bugs. For instance, a freed pointer variable is more critical than an arithmetic variable, as it could potentially lead to null pointer reference bugs in the suffix code logic of the program. Additionally, if we treat the data-suffixes with different levels of importance, the bug discovery process will be enhanced. Semantic information plays a significant role in determining the relevance of data dependencies. Factors such as data flow to a global value or memory write rather than memory reads carry more significance.

To enhance the precision and value of affected code targeting for vulnerability discovery in the commit-introducing changes, WAFLGo can collect and learn from the semantic information of the codes like [22]. By incorporating this semantic knowledge, one can achieve more accurate identification of affected code, ultimately improving the efficacy of vulnerability detection.

Incremental Testing. Different from traditional greybox fuzzing approaches, WAFLGo focuses on newly submitted commits to uncover newly introduced bugs. The ultimate goal is to shift commit security testing left in the Continuous Integration/Continuous Deployment (CI/CD) pipeline, ensuring that bugs are detected before a commit is merged into the main branch and released to the public. While our evaluation results demonstrate that we can discover a significant number of bugs within two hours, there are still instances where bug detection requires more than 12 hours (e.g., No. 5). This extended duration may not be feasible or acceptable in practical commit testing scenarios.

To address this challenge, an intuitive solution is to explore incremental testing. We could leverage historical testing data to guide WAFLGo more efficiently to the change site, similar in [25, 29]. Additionally, program analysis techniques such as under-constraint symbolic execution [6, 28, 34] and program slicing [9] can be utilized to further optimize the testing process. We consider these topics as future work, with the potential to enhance the efficiency and effectiveness of WAFLGo in the context of commit testing.

8 Related Work

In this section, we discuss the closely related works.

Improvement of Directed Greybox Fuzzing. Many approaches have been proposed to optimize directed greybox fuzzing and improve its effectiveness in discovering program bugs. Inspired by distance-guided fuzzer AFLGo [5], Hawkeye [7] introduces augmented distance metrics to enhance directness, while Windranger emphasizes precise distance calculation through the deviation of basic blocks. Reachability-based fuzzers like FuzzGuard [40] train models to predict and discard inputs that cannot reach the target, while Beacon [14] stops inputs that cannot reach the target. SelectFuzz [23] selectively instruments relevant code of the target to explore the program more efficiently. Parmesan [27] and FishFuzz [38] address the multi-target problem, while their targets are sanitizer-marked locations.

Different from existing works primarily focusing on quickly reaching the target, WAFLGo aims to thoroughly test the commit change sites and their affected code. Recognizing the importance of the commit's impact on the entire program, WAFLGo goes beyond reaching the target commit change sites and extensively tests the affected code with diverse inputs. By combining targeted testing with a comprehensive exploration of affected code, WAFLGo enhances its ability to uncover vulnerabilities introduced by code changes and contributes to more effective bug detection.

Verifying New Code. When a new commit is submitted to a program, it becomes crucial to verify whether this commit has introduced any new vulnerabilities. In the defect prediction community, lots of approaches [32–34] have been proposed to address this concern. For instance, UC-KLEE [28] utilizes under-constrained symbolic execution to directly check individual functions for potential crashes and to verify the new code. Symbolic execution provides a high level of certainty in code verification as it exhaustively explores the input space. However, it can be computationally intensive.

Regarding the fuzzing scope, AFLChurn [39] proposes simultaneous testing of all commits, assigning more energy to the inputs that execute the recent modified code area. In this work, we adopt a similar objective of targeting every commit change and addressing the multi-target challenge carefully. The aim is to identify newly introduced bugs as early as possible during the development process.

9 Conclusion

This work addresses the challenges faced in testing commits to discover newly introduced bugs and proposes WAFLGo, a directed fuzzer designed to efficiently reach change sites and thoroughly test their affected code. WAFLGo employs a critical code guided input generation strategy and a multi-target distance metric to enhance its effectiveness. We curate a dataset containing 30 real-world bugs, and identify their BICs manually. Through a meticulous evaluation of this dataset, we demonstrate the effectiveness of WAFLGo compared to existing fuzzers. Furthermore, WAFLGo successfully identifies and reports seven new vulnerabilities, including four CVEs, in real-world programs, showcasing its practical applicability.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their detiled and valuable comments. This work was partly supported by NSFC under No.62302443, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), Jianghuai Advance Technology Center under No. 00QK0021 and the Key R&D Program of Ningbo under Grant No.2023Z235.

References

[1] American Fuzzy Lop. https://github.com/google/AFL, 2021.

[2] libtiff issue #519. https://gitlab.com/libtiff/libtiff/-/issues/519, 2021.

[3] fig2dev issue #142. https://sourceforge.net/p/mcj/tickets/142/, 2022.

[4] OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2023.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[8] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. {SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis. In *USENIX Security Symposium*, pages 2531–2548, 2022.

[9] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 485–498, 2022.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[11] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2440–2451, 2022.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.

[13] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium*, pages 2577–2594, 2020.

[14] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[15] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. Darwin: Survival of the fittest fuzzing mutators. *arXiv preprint arXiv:2210.11783*, 2022.

[16] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[17] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *USENIX Security Symposium*, pages 3559–3576, 2021.

[18] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[19] Yuwei Li, Yuan Chen, Shouling Ji, Xuhong Zhang, Guanglu Yan, Alex X Liu, Chunming Wu, Zulie Pan, and Peng Lin. G-fuzz: A directed fuzzing framework for gvisor. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[20] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security Symposium*, pages 2777–2794, 2021.

[21] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2022.

[22] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences. In *USENIX Security Symposium*, pages 1769–1786, 2019.

[23] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1050–1064. IEEE Computer Society, 2022.

[24] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *USENIX Security Symposium*, pages 1949–1966, 2019.

[25] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *29rd Annual Network and Distributed System Security Symposium, NDSS*, pages 24–28, 2022.

[26] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. Slime: program-sensitive energy allocation for fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 365–377, 2022.

[27] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium*, pages 2289–2306, 2020.

[28] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, pages 49–64, 2015.

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[30] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2595–2609, 2022.

[31] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[32] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3282–3299, 2021.

[33] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. {MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components. In *USENIX Security Symposium*, pages 3037–3053, 2022.

[34] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[35] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[36] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium*, pages 2307–2324, 2020.

[37] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, pages 745–761, 2018.

[38] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. Fishfuzz: catch deeper bugs by throwing larger nets. In *USENIX Security Symposium*, pages 1343–1360, 2023.

[39] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

[40] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium*, pages 2255–2269, 2020.

Appendix

A Bug Information

For each bug used in our paper, we present the manually identified bug-inducing commit along with the corresponding bug-fixing commit in Table 5.

Table 5: Bug information. "BIC Hash" and "Fix Hash" represent the bug-inducing commit hash and the bug-fixing commit hash, respectively. "#Changed lines" refers to the number of modified lines in the commit.

Project	Program	Issue	BIC Hash	#Changed lines	Fix Hash	#Changed lines
Libtiff	tiffcrop	#488	7057734d	40+,17-	97d65859	1+,1-
	tiffcrop	#498	07d79fcac	51+,26-	82a7fbb1	66+, 2-
	tiffcrop	#519	f13cf46b	9+,2-	69818e2f	35+, 24-
	tiffcrop	#520	e3195080	210+,72-	6366e8f7	53+, 19-
	tiffcrop	#527	07d79fcac	51+,26-	ec8ef90c	13+, 34-
	tiffcrop	#530	f13cf46b	9+,2-	b0e1c25d	7+, 0-
	tiffcp	#548	3079627e	244+,137-	9be22b63	5+, 0-
	tiffinfo	#559	b90b20d3	1647+,1538-	e8874d75	1879+, 22-
Bento4	mp4info	#652	c9f2c53	33+,18-	8d7253e	1+, 1-
	mp4info	#679	2e29350	1148+,742-	902210c	29+, 25-
	mp4audioclip	#732	bbb6f24	1045+,1688-	df9ba99	6+, 1-
	mp42aac	#751	61b2012	0+,6-	1565b65	10+, 4-
Mujs	mujs	#65	8c27b126	27+,16-	833f82c	2+, 0-
	mujs	#141	832e0690	87+,27-	6871e5b	6+, 0-
	mujs	#145	4c7f6be	41+,5-	9c76d8e	1+, 1-
	mujs	#166	3f71a1c9	260+,47-	8b5ba20	17+, 47-
Libjpeg	cjpeg	#493	88ae609	1999+,228-	1719d12	15+, 2-
	jpegtran	#636	88ae609	1999+,228-	dc4a93f	5+, 1-
Tcpreplay	tcprewrite	#702	0a65668a	282+,148-	c23738f	3+, 2-
	tcprewrite	#718	2c76868d	45+,45-	ad346b7	89+, 77-
	tcpprep	#756	16442ac3	312+,338-	00b6601	1+, 2-
	tcpreplay	#772	4f9158da	1+,2-	5c59132	19+, 2-
Libxml2	xmllint	#535	9a82b94a	253+,176-	d0c3f01e	0+, 2-
	xmllint	#550	7e3f469b	32+,38-	6273df6c	6+, 5-
Poppler	pdfunite	#1282	3d35d209	16+,0-	0b9f7022	4+, 0-
	pdfunite	#1289	3cae7773	31+,2-	efb68686	13+, 2-
	pdftops	#1303	e674ca64	71+,80-	a4ca3a96	4+, 0-
	pdftoppm	#1305	aaf2e808	31+,2-	907d05a6	1+, 1-
	pdftoppm	#1381	245abada	20+,45-	1be35ee8	22+, 20-
ImageMagick	magick	#6075	a107b941	103+,134-	8c97870	3+, 0-