Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 7698916, 12 pages
https://doi.org/10.1155/2020/7698916

Research Article

RDFuzz: Accelerating Directed Fuzzing with Intertwined Schedule and Optimized Mutation

Jiaxi Ye, Ruilin Li, and Bin Zhang

College of Electronic Science, National University of Defense Technology, Changsha, China

Correspondence should be addressed to Ruilin Li; securitylrl@gmail.com

Received 12 September 2019; Revised 13 January 2020; Accepted 20 February 2020; Published 17 March 2020

Academic Editor: Jorge Rivera

Directed fuzzing is a practical technique, which concentrates its testing energy on the process toward the target code areas, while costing little on other unconcerned components. It is a promising way to make better use of available resources, especially in testing large-scale programs. However, by observing the state-of-the-art-directed fuzzing engine (AFLGo), we argue that there are two universal limitations, the balance problem between the exploration and the exploitation and the blindness in mutation toward the target code areas. In this paper, we present a new prototype `RDFuzz` to address these two limitations. In `RDFuzz`, we first introduce the *frequency-guided strategy* in the exploration and improve its accuracy by adopting the branch-level instead of the path-level frequency. Then, we introduce the *input-distance*-based evaluation strategy in the exploitation stage and present an optimized mutation to distinguish and protect the distance sensitive input content. Moreover, an intertwined testing schedule is leveraged to perform the exploration and exploitation in turn. We test `RDFuzz` on 7 benchmarks, and the experimental results demonstrate that `RDFuzz` is skilled at driving the program toward the target code areas, and it is not easily stuck by the balance problem of the exploration and the exploitation.

1. Introduction

The enormous scale in modern software makes it a difficult task to conduct a thorough testing within a limited time budget and computing resources. Furthermore, in many practical scenarios, only some code areas need testing, such as the patches, security-sensitive functions, or some user-defined positions [1]. At present, the directed testing techniques are promising solutions to satisfy this requirement.

The state-of-the-art directed testing techniques mainly include *fuzzing-based* groups and *symbolic execution-based* groups [1–4]. However, owing to some unsolved limitations, e.g., the path explosion problem, the application of the symbolic execution [5] is still narrow on the large-scale software. On the contrary, the fuzzing techniques are advantaged at testing these large-scale programs, and they have successfully uncovered a lot of vulnerabilities so far.

The study on the directed fuzzing techniques is currently extensive. More specifically, fuzzing is generally regarded as a random process, so that the directed fuzzing can be modeled as an optimal search, starting from an arbitrary input and searching for the inputs that can hit target code areas. Researchers have introduced some metaheuristic approaches to implement such searching and improve its performance. For example, Böhme et al. [1] presented a directed fuzzing tool, AFLGo. It leverages a distance-based evaluation to distinguish the inputs remote to the target code areas, and it uses a simulated annealing- (SA-) based schedule strategy to distribute the testing energy by the `input-distance` evaluation. In brief, it can steer the testing engine toward the target code areas, and its experiments show a better directed performance than the *symbolic execution-based* methods.

We ran AFLGo (directed fuzzing) and AFL (nondirected fuzzing) for certain times and examined their results. In some runs, AFLGo presents a faster speed than AFL to the target code areas; however, in other runs, the directed performance by AFLGo is not as expected to outperform AFL. This experimental result is caused by the different user-

defined configurations in AFLGo runs. More specifically, there are two running stages in AFLGo, the *exploration* and *exploitation* stage. The *exploration* stage is designed to uncover as many codes as possible; then based on the revealed results, the *exploitation* stage is invoked to drive the engine to the target code areas. AFLGo adopts a timewise splitting method to coordinate these two stages; i.e., it first runs the *exploration* stage and then runs the *exploitation* stage, and the user-defined parameters settle the time budgets for these two stages. Because an inadequate *exploration* result usually leads to a poor *exploitation* result, AFLGo did not outperform AFL in some runs.

Besides, the random mutation is a comprehensive strategy in exploration stage to discover code areas. Nevertheless, we argue that the random mutation is weak in the directed testing, because it is blindly driving the program toward the target code areas, and it tends to destroy some pivot input content, violating the directing process.

In this paper, as a framework, we introduce the *frequency guided strategy* [6] in the exploration and the *input-distance*-based evaluation strategy [1] in the exploitation. More specifically, we first improve the exploration by utilizing the branch-level instead of the path-level frequency. Then, in the exploitation, we argue that every input content plays a different role in the process toward the target code areas; for example, some input contents are just the carriers of data, whereas some other input contents can significantly affect the program execution. Therefore, we present a *disturb-and-check* method to identify and protect the distance sensitive input content, that is, our optimized mutation, in an attempt to produce inputs closer to the target code areas. During testing, the opportunity to invoke the exploration/exploitation stage is determined by the input evaluation results; the engine starts to explore when the input is helpful to uncover the codes, and it starts to exploit when the input is helpful to the target code areas; this is our intertwined testing schedule, an approach to handle the balance problem between the exploration and the exploitation.

On the basis of the techniques, we develop a prototype, dubbed by `RDFuzz`. In the experiments, we deploy `RDFuzz` on 7 benchmarks, investigating the effectiveness of our optimized mutation and the directed performance of `RDFuzz`. The results demonstrate that our techniques are useful to speed up the testing toward the target code areas.

The rest of this paper is organized as follows. In Section 2, we present an overview of some background knowledge, the problems in directed fuzzing, and the framework of our solution in brief. Section 3 describes our proposed techniques in detail. Section 4 reports the implementation and presents experimental results. Section 5 elicits the threats to validity, and Section 6 is about our future work. Our conclusion is in Section 7.

2. Overview

2.1. Fuzzing Algorithm. Since its introduction by Miller et al. in 1990 [7], fuzzing has been the most useful technique for vulnerability detection [8]. It continuously feeds the program under testing with numerous inputs, which are generated by different strategy-oriented methods, in an attempt to find bugs in the program.

Algorithm 1 provides a generic fuzzing process. It first reads an initial seed pool ($\mathcal{SP}$), which can be randomly generated or crawled from the Internet [9]; then, it activates a testing iteration and terminates when the `Continue()` function returns *False*. In each testing round, on the basis of different evaluation principles, the engine selects a basal input from $\mathcal{SP}$ via the `Choose()` function and produces a new input s' by mutation on s, the program executes on input $s\prime$, and the `BugOracle()` function inspects the bug(s) on the basis of the execution information (`info`); thereafter, the seed pool is updated in `SPUpdate()` function, and the engine would launch the next testing round.

In this fuzzing algorithm, the `Choose()` and `Mutation()` functions are two significant parts. The `Choose()` function determines how to pick suitable inputs for mutation, that is, the starting line of the searching, and the `Mutation()` function determines how to produce new inputs, that is, the searching process. As far as we know, most enhanced fuzzing techniques are proposed through researching on these two parts.

2.2. Exploitation and Exploration Strategies in Fuzzing. There are many approaches for fuzzing taxonomy, such as black-box, grey-box, and white-box fuzzers [8] and mutation-based and generation-based fuzzers [10]. To a certain point, the fuzzing techniques can be classified into *exploitation* and *exploration* techniques.

Exploration strategy is an attempt to perform thorough testing on the program. More specifically, many fuzzers take the testing coverage as an indication of the testing degree [11]. In order to maximize the testing coverage, a lot of new approaches are put forward, and they yield a lot of new fuzzers, like AFLFast [6], FairFuzz [12], Angora [13], VUzzer [14], and CollAFL [15].

On the contrary, *exploitation* strategy is an attempt to perform concentrated testing on parts of the program. Because modern software is almost huge, it is difficult to explore the whole software in a restricted time budget or by limited computing resources. Moreover, in many scenarios, thorough testing on the target program is not necessary; only some code areas demand testing, e.g., the program patches and sensitive functions.

2.3. Directed Fuzzing and Its Problems. In the fuzzing community, the *exploitation* strategy is often leveraged in the *directed fuzzing*. It makes efforts to focus the testing on the process toward the target code areas and reduce the executions on the nontarget regions. More specifically, this mechanism is implemented by the modification on the `Choose()` and `Mutation()` functions (Algorithm 1). The `Choose()` function would pick out the inputs close to the target code areas; based on these inputs, the `Mutation()` function makes special manipulations to produce new inputs, in an attempt to steer the program toward the target code areas.

```
    Input: Seed pool (𝒮𝒫)
    Output: ℬ𝓊𝑔
(1) ℬ𝓊𝑔 ⟵ ∅;
(2) Function Fuzzing():
(3)   while s ⟵ Choose 𝒮𝒫 ∧ Continue() do
(4)     s′ ⟵ Mutation(s);
(5)     info ⟵ Execute(s′);
(6)     bug ⟵ BugOracle(info);
(7)     ℬ𝓊𝑔 ⟵ ℬ𝓊𝑔 ∪ bug;
(8)     𝒮𝒫 ⟵ SPUpdate(info);
(9)   end
(10) end
```

Algorithm 1: Fuzzing algorithm.

Furthermore, the *exploitation* is usually accompanied by the *exploration*, because exploration is to gather information and exploitation is to make decisions depending on the gathered information. In other words, the fuzzer adopts the exploration to uncover more code areas and the exploitation to find the positions closest to (or at) the target code areas.

For example, the directed fuzzer AFLGo [1] performs the *exploitation* strategy as follows: its `Choose()` function leverages an `input-distance`-based approach to determine the better and worse inputs, and its `Mutation()` function introduces a simulated annealing- (SA-) based approach to assign more testing energy to the better inputs and less to the worse inputs. Moreover, it employs a timewise splitting method to coordinate the exploitation and the exploration, it uses a "$-z$" parameter defined by the user to set a fixed time budget for the exploration, and the engine invokes the *exploitation* stage in the remaining time.

To assess the cooperation of *exploitation* and *exploration* strategies in directed fuzzing, we examine how AFLGo works with different "$-z$" parameters on the same benchmark. The experiments last for 24 hours; `AFLGo-1` means 1 hour of exploration and 23 hours of exploitation, and the other configurations are shown in Table 1. The "min distance" (Y label in Figure 1) means the minimum `input-distance` among all the generated inputs to the target code areas. A lower "min distance" denotes a better directed performance, explained in detail in Section 4.

In Figure 1, the comparing result of the directed performance is `AFLGo-8` > `AFLGo-4` > `AFLGo-2` > `AFLGo-1`. It is caused by their different time budgets for the exploration. Compared to `AFLGo-1`, which performs an hour of exploration, `AFLGo-8` sustains 8 hours of exploration, so that `AFLGo-8` has gathered sufficient information for the following exploitation stage; i.e., the engine has uncovered a certain amount of the code areas; thus, `AFLGo-8` outperforms `AFLGo-1` in the directed performance as a result.

2.3.1. Tradeoff Problem. We argue that it is a universal tradeoff problem to coordinate exploration and exploitation in the directed fuzzing. On the one hand, more exploration can obtain and provide adequate information for the exploitation; on the other hand, an overfull exploration would occupy many resources, and the exploitation is delayed as a result. AFLGo only employs a user-defined parameter to settle the time budgets for exploration and exploitation; this is maladaptive, and it would bring in some unexpected results. Figure 2 shows the tradeoff problem in AFLGo.

In particular, Hawkeye [2], including other improved directed fuzzing engines, is enhanced by a stronger exploitation ability to reach target code areas. However, the tradeoff problem still constitutes a bottleneck in their performance.

2.3.2. Mutation Problem. Besides, in the directed fuzzing, the mutation functions should conduct some special operations to adjust the input generation to speed up the directed process. However, the random mutation used brings in a severe limitation; due to the blindness, the following mutation would destroy some critical input contents previously generated. It causes deterioration in the input quality so that the directed fuzzing performance is limited.

2.3.3. Summary. To conclude, we summarize the two limitations in the present directed fuzzing techniques as follows: the tradeoff problem between the exploitation and the exploration and the blindness in mutation.

2.4. Framework of Our Approach. To address the limitations mentioned above, we present a new directed fuzzing tool, dubbed by `RDFuzz`, and Figure 3 shows its framework. There are two testing loops: (i) the exploration loop is designed to improve the code coverage and provide a sufficient discovery; (ii) the exploitation loop conducts the searching for the inputs on the target code areas.

There are three significant parts in the framework (marked with green). The input evaluation appraises each input, determines its input type, schemes the testing energy, and decides which testing loop to invoke (exploration/exploitation), to make excellent use of the inputs from the seed pool. The distance sensitive content is distinguished by a *disturb-and-check* method, which relies on the

Table 1: The time budgets for the exploration and exploitation.

	Time (hours)	
	Exploration	Exploitation
AFLGo-1	1	23
AFLGo-2	2	22
AFLGo-4	4	20
AFLGo-8	8	16

`input-distance` calculation. The optimized mutation is aware of the distance sensitive content, and it protects these input contents from variation in producing new inputs.

3. Methodology

This section presents our improvements on the exploration strategy in 3.1 and the improvements on the exploitation strategy in 3.2 and describes the scheduled workflow in 3.3.

3.1. Improvements on the Exploration Strategy. It is common knowledge that an abundant exploration result empirically leads to a better global search capability and a better ability to avoid falling into the local optimum. To achieve such a result in the directed fuzzing, we introduce the *frequency guided strategy* [6], which is aimed at discovering more code areas, especially the codes deeply buried in the program.

In this strategy, the execution frequency is counted, and the code areas are separated into *high-frequency* and *low-frequency* areas according to a certain threshold. Based on the Markov model [16], the *high-frequency* code areas are regarded as easily accessible, and the *low-frequency* positions are difficultly accessible. Therefore, the testing energy distribution should be lean to the *low-frequency* positions, in an attempt to uncover more codes.

The original *frequency guided strategy* is conducted on the path-level, we meliorate its accuracy by figuring out the *high-frequency*/*low-frequency* code areas on the branch-level statistics. Two reasons inspire this modification.

The first reason is that the path-level statistic may misinform the *low-frequency* code areas. We take the example in Figure 4 as an explanation. There are 4 paths and 4 branches (A, B, C, D, and E are five basic blocks). It is assumed that path $\mathtt{B} \longrightarrow \mathtt{C} \longrightarrow \mathtt{E}$ is executed 5 times, and the other three paths are executed 100 times. On the path-level, the minimum execution frequency is only 5% of the maximum, so that the engine would regard the $\mathtt{B} \longrightarrow \mathtt{C} \longrightarrow \mathtt{E}$ path as the *low-frequency* code areas and regulate more testing energy on the inputs running along with $\mathtt{B} \longrightarrow \mathtt{C} \longrightarrow \mathtt{E}$ path. However, on the branch-level, the ratio between the minimum execution frequency and the maximum is 50.5%, which is not a remarkable value in the fuzzing process, so that none of these 4 branches would be regarded as the *low-frequency* code areas, and the testing energy on these branches is limited in a certain range.

The second reason is that the path is a kind of sequential structure, and it is expensive to restore the sequential structures in the fuzzing process. To make a balance between efficiency and effectiveness, Zalewski [17] presents an approach of splitting a sequential path into an unordered group of branches. Besides, the original *frequency guided strategy* chooses to restore parts of the discovered paths, which are selected by a corpus distillation technique [18].

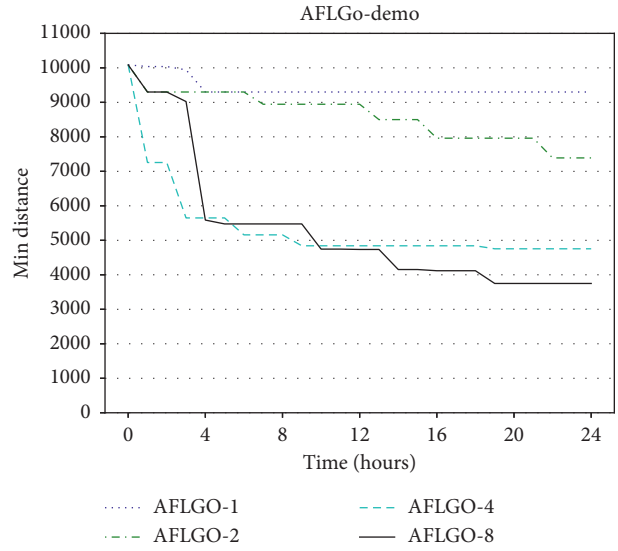


Figure 1: The directed fuzzing results. A lower "min distance" denotes a better directed performance.

According to the analysis above, we prefer to determine the *high-frequency*/*low-frequency* code areas according to the branch-level statistic. Then, the inputs running on the *low-frequency* code areas would be prioritized and obtain superiority in testing energy distribution.

The execution number of each branch is counted during the testing, to make the branch-level statistics. It is realized by a global counter and some instrumentation. The counter maintains the total execution number of each branch, and the instrumentation informs the counter to add a certain number after executing a branch. After that, the testing energy on each input is regulated by the minimum execution number of all the branches in its trace.

Formally, for a branch `br`, its execution number `numBr[br]` is accumulated as equation (1). $\mathscr{R}$ is the record of all the executed inputs, and `NumHit(br, s)` is the execution number of branch `br` on running input s. After obtaining the statistic, for an input s, its frequency $F_{\text{br}}(s)$ is chosen as the minimum branch-level frequency on its execution trace, as equation (2) shows, where $T(s)$ denotes a set of all the appearing branches by running input `s`.

$$\text{numBr}[\text{br}] = \sum_{s \in \mathscr{R}} \text{NumHit}(\text{br}, s), \tag{1}$$

$$F_{\text{br}}(s) = \min_{\text{br} \in T_{\text{br}}(s)} [\text{numBr}[\text{br}]]. \tag{2}$$

Furthermore, Algorithm 2 describes a general process of the exploration stage. It iterates the inputs in the seed pool (Ln.1–Ln.6); for each selected input, the engine calculates its minimum branch-level frequency $F_{\text{br}}(s)$ (Ln.2) and assigns a suitable testing energy $E(s)$ to it (Ln. 4); then it invokes the random input generation to perform the testing (Ln.5).

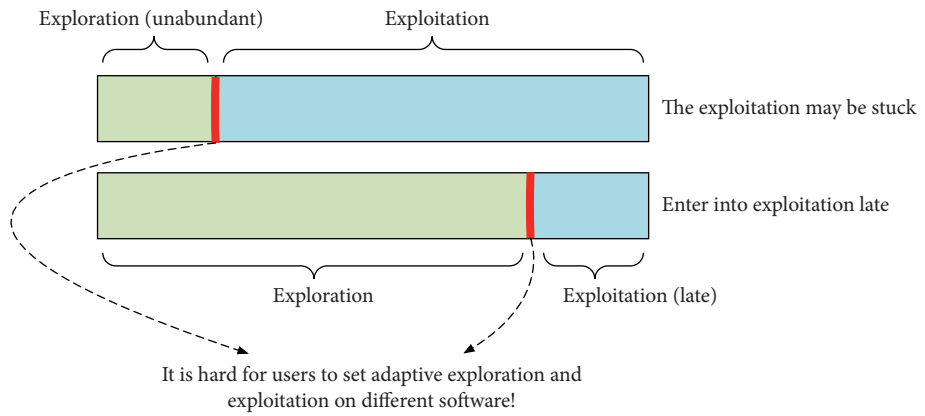


Figure 2: The exploration and exploitation tradeoff problem in AFLGo.

In practice, there are many feasible solutions in `GetEnergyByFreq()` function; the distribution can be exponential, linear, quadratic [19], etc.

3.2. Improvements on the Exploitation Strategy. The exploitation is to drive the program toward the target code areas, and it is conducted based on the exploration results. Our exploitation strategy consists of an `input-distance` calculation and an optimized mutation. The former is introduced from AFLGo [1], and it can determine whether an input is close to the target codes or not. The latter attempts to avoid producing inputs whose distances are farther than the original input for mutation.

3.2.1. Distance Calculation. The `input-distance` is calculated by five steps, indicating the distance from the input execution trace to the target code areas.

(i) The call graph (CG) and control flow graph (CFG) are extracted from the target program by static analysis. It can be done by the functions provided by LLVM [20] on the program source codes.

(ii) On the CG, supposing the target code areas are located in the f_T function, the distance between an arbitrary function to the f_T function can be obtained by some certain graph-based algorithms [21]. This function level distance is represented as $d_{ff}(f_i, f_T)$.

(iii) On the CFG, for any two arbitrary basic blocks, their distance can also be calculated by some certain graph-based algorithms. This basic block level distance is represented as $d_{bb}(b_i, b_j)$.

(iv) Based on the function and basic block level distances, the distance from an arbitrary basic block b_i to the target function f_T is calculated and represented as $d_{bf}(b_i, f_T)$. More specifically, to compute this distance, it should locate a pivotal basic block b_k, which acts as a middle position to compose a feasible path $b_i \longrightarrow b_k \longrightarrow f_T$. The b_k basic block contains a `call` instruction, which joints the basic blocks (in CFG) and functions (in CG). In particular, if there are multiple choices for b_k, the distance is settled as an average value.

The calculation is shown in (3), where $\mathcal{S}$ denotes a set of the feasible b_k choices, b_k calls f_k, and α is a constant parameter.

$$d_{bf}(b_i, f_T) = \text{average} \times \left(\sum_{\substack{b_k \in \mathcal{S} \\ b_k \longrightarrow f_k}} \left[d_{bb}(b_i, b_k) + \alpha \cdot d_{ff}(f_k, f_T) \right] \right). \tag{3}$$

(v) For each input, there is a series of basic blocks in its trace, and each basic block is assigned a distance value in step iv, so that the `input-distance` can be calculated by the distances of the basic blocks emerging in the input execution trace.

The calculation is formally shown in (4). $T_b(s)$ is a set of all the basic blocks in the execution trace of input `s`; $d_{bf}(b_{i1}, f_T)$ means the distance from basic block b_{i1} to the target function f_T; F function reads all the distances of the basic blocks, which are in the $T_b(s)$, and outputs a final value as the `input-distance` $d_s(s, f_T)$.

$$d_s(s, f_T) = F\left(d_{bf}(b_{i1}, f_T), \ldots, d_{bf}(b_{in}, f_T)\right) \Big| \, b_{i1}, \ldots, b_{in} \in T_b(s). \tag{4}$$

We take an example to describe the distance calculation from a basic block to the target code areas (step iv). In Figure 5, b_1 is the considered basic block, and the target code areas are in the f_3 function. According to the CFG and CG, there is a reachable path from b_1 to f_3, i.e., $b_1 \longrightarrow b_2 \longrightarrow b_3 \longrightarrow f_1 \longrightarrow f_2 \longrightarrow f_3$, so that the distance from b_1 to f_3 is calculated as (5) shows, α is a constant parameter.

$$d_{bf}(b_1, f_3) = d_{bb}(b_1, b_3) + \alpha \cdot d_{ff}(f_1, f_3) \tag{5}$$

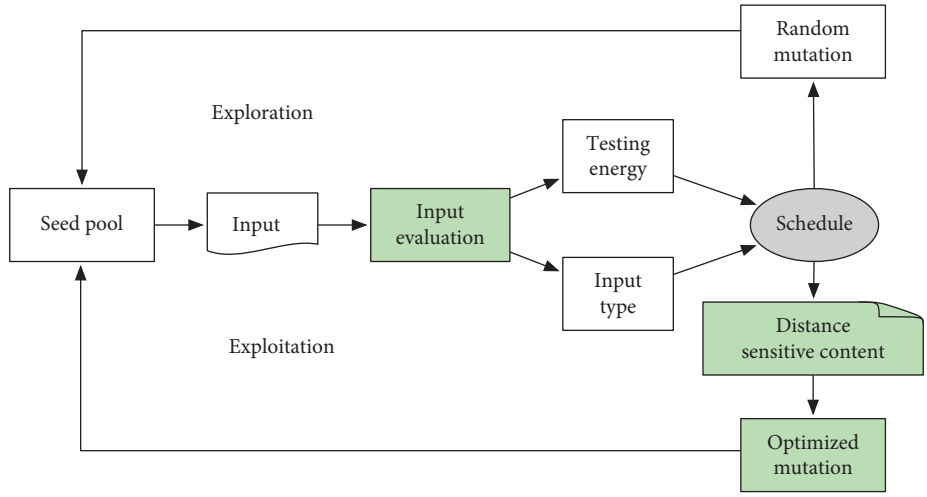
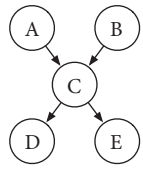


Figure 3: The framework of RDFuzz.



Path execution num:	Branch execution num:
A → C → D : 100	A → C : 200
A → C → E : 100	B → C : 105
B → C → D : 100	C → D : 200
B → C → E : 5	C → E : 205

Figure 4: An example of the execution frequency.

```
(1) for s in Seed Pool (SP) do
(2)     F(s) ⟵ GetFrequency(s);
(3)     /* Assign testing energy by the exploration schedule */
(4)     E(s) ⟵ GetEnergyByFreq(F(s));
(5)     RandomMutation(s, E(s));
(6) end
```

Algorithm 2: The exploration process.

3.2.2. Optimized Mutation. Based on the `input-distance` calculation, we further propose an optimized mutation to help the engine reach the target code areas. The core idea is to prevent the input generation from destroying the crucial input content.

The concept of crucial input content is widely employed in the fuzzing research; however, it has various meanings. In our research, it particularly refers to the distance sensitive content. The distance sensitive feature indicates that the content is vital to maintain the `input-distance`; once the content is altered, the `input-distance` would become larger; i.e., the input becomes farther from the target code areas. As a typical configuration, the input content is represented in a bytewise granularity in this paper.

We design a *disturb-and-check* method to detect the distance sensitive bytes. The method is implemented based on the `input-distance` comparison between the original input and the mutated input. Given an original input s_o, a byte content is altered (assuming the `No. k` byte) and a new input s_k is produced. By comparing the `input-distance` of s_o and s_k, it can make an assertion about whether the `No. k` byte is distance sensitive or not. After traversing the input, it can obtain all the distance sensitive bytes.

In the directed fuzzing, the goal of the engine is to continuously produce the inputs that are close to the target code areas, until reaching them. Therefore, we argue that it is beneficial to protect the distance sensitive bytes in the input generation. By adopting this strategy, the input generation is optimized and called `OptimizedMutation`.

In addition, we take an example to explain why the distance sensitive input bytes should be protected. Figure 6 shows a code snippet; variables `x` and `y` are from the input, supposing that `x` is from the `5th` and `y` is from the `6th` input byte. There is a bug at Ln.6, i.e., the `abort()` function, and it is settled as the target code. To ensure a more clear description, we take the code line as a representation to describe the basic block level distance $d_{bf}(b_i, f_T)$. Because Ln.6 is the target code, its distance is 0, and the distances of the other code lines are shown in Table 2.

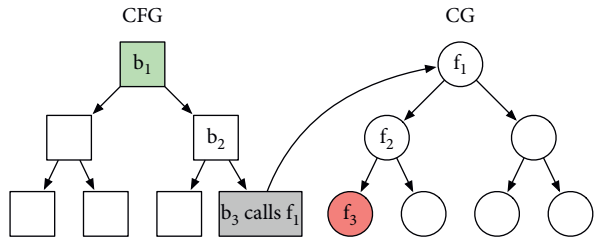


Figure 5: An example to show the distance computation from a basic block to the target code areas.

```
1 void check(){
2   x=input(); //from 5th byte
3   y=input(); //from 6th byte
4   if(x==100)
5     if(y==200)
6       abort(); //target code
7 }
```

Figure 6: A code snippet in the directed fuzzing.

Table 2: The distance of each code line.

Line	Distance
Ln.2	4
Ln.3	3
Ln.4	2
Ln.5	1
Ln.6	0

Given an input s, assuming `x` is 100 and `y` is 150 on the execution trace, the program can execute till Ln.5, and its `input-distance` is $(4 + 3 + 2 + 1/4) = 2.5$. Once the `5th` byte is altered, and a new input s_n is generated, `x` is not 100 anymore, the program only executes till Ln.4, and the `input-distance` of s_n is $(4 + 3 + 2/3) = 3$. As the `input-distance` becomes larger, the `5th` input byte is distance sensitive. In the mutation, if the `5th` input byte is protected, `x` can be kept as 100, and the program can always execute till Ln.5, increasing the possibility of reaching Ln.6 (the target code).

Finally, Algorithm 3 describes a generic process of the exploitation. It iterates the inputs in the seed pool (Ln.1–Ln.5); for each selected input, the engine calculates its `input-distance` (Ln.2) and assigns a suitable testing energy to it (Ln.3); then it invokes distance sensitive optimized mutation to perform the testing (Ln.4).

The practical solutions in `GetEnergyByDis()` function can be heuristic; for example, AFLGo [1] leverages a simulated annealing approach to assign the testing energy, and Hawkeye [2] leverages a balanced power function to regulate the testing energy.

3.3. Intertwined Schedule for Directed Testing. According to the techniques mentioned above, based on the evaluation results about the frequency and `input-distance`, the inputs can be classified into `HD/LD` (high/low distance) and `HF/LF` (high/low frequency) types. By grouping the two evaluation criteria, the inputs can be classified into four types. They are shown in Table 3.

Because the `LF` inputs are helpful to improve the coverage, they are required in the exploration; the `LD` inputs are helpful to achieve the target code areas; they are required in the exploitation. Different from the timewise splitting schedule [1, 2], an intertwined testing schedule is designed in the directed testing, as Algorithm 4 shows. In a nutshell, it is a combination to alternately conduct the exploration (Algorithm 2) and exploitation (Algorithm 3);

(1) **for** s *in Seed Pool* ($\mathcal{SP}$) **do**
(2) $D(s) \longleftarrow$ GetDistance(s);
(3) $E(s) \longleftarrow$ GetEnergyByDis($D(s)$);
(4) OptimizedMutation(s, $E(s)$)
(5) **end**

Algorithm 3: The exploitation process.

moreover, the testing energy is assigned according to their frequency and `input-distance`.

In total, this schedule manifests a self-evolution characteristic, and it can alternately improve the exploration results and the exploitation results, in an attempt to avoid being stuck into the local optimum in the directed fuzzing process.

4. Implementation and Evaluation

Based on the proposed methods, we develop a prototype, dubbed by RDFuzz that is established on AFL [22], LLVM-based analysis [20], and a python script with the NetworkX package. AFL provides a fundamental testing framework; besides, we add some extra codes into *afl-fuzz.c* module, implementing our proposed methods, including the frequency statistics, the `OptimizedMutation()` function, and the testing energy distribution. LLVM-based analysis and the python script are employed in the `input-distance` calculation.

4.1. Experimental Infrastructure. All experiments were conducted on a server equipped with Intel Xeon CPU E5-2620 v2 @ 2.10GH (12 cores in total) and 64 GB RAM, running Linux Ubuntu 14.04 LTS AMD64. All the experiments were launched in single mode, with the same initial seeds.

Table 3: The classification of the inputs.

		Frequency	
		High frequency (HF)	Low frequency (LF)
Distance	High distance (HD)	HFHD	LFHD
	Low distance (LD)	HFLD	LFLD

```
(1) for s in Seed Pool (SP) do
(2)    F(s) ⟵ GetFrequency(s);
(3)    D(s) ⟵ GetDistance(s);
(4)    E_f(s) ⟵ GetEnergyByFreq(F(s));
(5)    E_d(s) ⟵ GetEnergyByDis(D(s));
(6)    Type(s) ⟵ Classify(D(s), F(s));
(7)    switch Type(s) do
(8)       case HFLD do
(9)          OptimizedMutation(s, E_d(s));
(10)      end
(11)      case LFHD do
(12)         RandomMutation(s, E_f(s));
(13)      end
(14)      Case LFLD do
(15)         OptimizedMutation(s, E_d(s) + E_f(s));
(16)      end
(17)      Otherwise do
(18)         continue;
(19)      end
(20)   end
(21) end
```

Algorithm 4: The intertwined schedule.

4.2. Benchmark. In the experiments, we selected 7 benchmarks to test and verify our proposed techniques; the information is shown in Table 4. Specifically, the first benchmark is according to the tutorial in AFLGo document [23], are the other benchmarks are the bug instances in the real-world programs (with CVE IDs).

4.3. Target Code Areas. The target code areas are set before testing. On the AFLGo-demo benchmark, the target code areas are set according to the document [23]. On the other benchmarks, the target code areas are set according to their crash traces. For example, Figure 7 shows the crash trace of CVE-2017-9050 (from ASAN report [24]), so that the target code areas are selected as `dict.c:285`, `dict.c:926`, and `parser.c:3425` (not all the codes in the trace are the target codes). All the ASAN reports of the benchmarks are taken from the web [25] or generated by executing the crash samples.

Though the target code areas are set according to the ASAN reports, all the benchmarks in our experiments are not compiled with addressSanitizer [26], because compiling with addressSanitizer is an augmented behavior for vulnerability detection, but it is not our main investigation point; in addition, it would inevitably lead to much overhead; in the experiments, a much larger time budget should be set to obtain a uniformly convincing evaluation result with addressSanitizer.

4.4. Research Questions. The following experiments were designed to answer two research questions:

(i) RQ1: Does the `OptimizedMutation` work as expected to improve the possibility of generating the inputs, which are closer to the target code areas than the original input?

(ii) RQ2: Does `RDFuzz` manifest a well directed performance in reaching the target code areas?

4.5. Evaluation of `OptimizedMutation` *(RQ1).* Due to the lack of the testing feedback, it is possible that the random mutation produces inputs same as or better/worse than the original input. On the contrary, the optimized mutation is designed in such a way to ensure the generation of inputs is closer to the target code areas.

In order to make a comparison between the random mutation and the optimized mutation, we present two appraisal indexes (`AI1`, `AI2`). The first index (`AI1`) is the proportion of the inputs holding `input-distance` not larger than the original input; i.e., $d_n \le d_o$, where d_n is the `input-distance` of the generated input and d_o is the `input-distance` of the original input. The second index (`AI2`) is the proportion of the inputs holding an `input-distance` smaller than the original input; i.e., $d_n \le \beta \cdot d_o$, where β is a constant smaller than 1, which we consider as 0.95 in the experiments.

More specifically, `AI1` is used to evaluate the ability to generate inputs which can at least keep their distances to the target code areas; i.e., the generated inputs do not go to other positions far from the target code areas. However, mutating on the noncritical bytes can also generate such inputs, yet it is a meaningless mutation. Therefore, we further introduce `AI2` to evaluate the ability to generate inputs that are closer than the original input to the target code areas.

Besides, as `RDFuzz` is built on AFL, which includes two mutation stages, `Det` and `Havoc` [27], we make evaluation on `Det` and `Havoc`, respectively. In `Det` stage, the input generation is dependent on a slight mutation, and in the `Havoc` stage, the engine leverages a heavy mutation for input generation.

The evaluation results w.r.t the `AI1` appraisal index are shown in Table 5. It can be seen that, for each benchmark under the same mutation, the proportion in `Det` is much larger than that in the `Havoc` stage. This is an expected result, because the inputs produced in `Det` have strong similarity to the original input (slight mutation), and the ones produced in `Havoc` have only weak similarity (heavy mutation).

For each benchmark, by comparing the proportions in `Det` and `Havoc` stages, respectively, it is concluded that the

Table 4: The information about the benchmarks.

	Library	Program	Version	Parameter
AFLGo-demo	libxml2	xmllint	commit ef709ce2	--valid--recover file
CVE-2017-9050	libxml2	xmllint	v2.9.4	--oldxml10 file
CVE-2015-7497	libxml2	xmllint	v2.9.2	--recover file
CVE-2015-8241	libxml2	xmllint	v2.9.2	file
CVE-2016-10094	libtiff	tiffsplit	4.0.7	file-o/dev/null
CVE-2015-4695	libwmf	wmf2svg	0.2.8	--wmf-fontdir = /dir/gsfonts file-o/dev/null
CVE-2018-19519	tcpdump	tcpdump	4.9.2	-ee-vv-nnr file

```
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x603000000030
 at pc 0x0000004c1685 bp 0x7ffc15d12290 sp 0x7ffc15d11a40
READ of size 109 at 0x603000000030 thread T0
#1 0xaf4b91 in xmlDictAddString /libxml2/dict.c:285:5
#2 0xaf4b91 in xmlDictLookup__internal_alias /libxml2/dict.c:926
#3 0x522740 in xmlParseNameComplex /libxml2/parser.c:3425
#4 0x522740 in xmlParseName /libxml2/parser.c:3487
#5 0x56afe6 in xmlParseElementDecl /libxml2/parser.c:6718:16
#6 0x56d4f0 in xmlParseMarkupDecl /libxml2/parser.c:6997:4
#7 0x5abe66 in xmlParseInternalSubset /libxml2/parser.c:8482:6
#8 0x5a9ec4 in xmlParseDocument /libxml2/parser.c:10930:6
#9 0x5d7bd8 in xmlDoRead /libxml2/parser.c:15445:5
#10 0x7fa6e05d7bd8 in xmlReadFile /libxml2/parser.c:15507
#11 0x521ac8 in parseAndPrintFile /libxml2/xmllint.c:2408:9
#12 0x51872d in main /libxml2/xmllint.c:3775:7
```

Figure 7: The ASAN report of CVE-2017-9050.

Table 5: AI1: the proportion of the inputs with no larger input-distance among all the mutated inputs.

Benchmark	Random mutation		Optimized mutation	
	Det (%)	Havoc (%)	Det (%)	Havoc (%)
AFLGo-demo	69	12	78	20
CVE-2017-9050	67	16	91	25
CVE-2015-7497	69	11	81	22
CVE-2015-8241	73	16	95	31
CVE-2016-10094	65	41	97	89
CVE-2015-4695	70	45	95	80
CVE-2018-19519	65	32	94	23
Average	68	25	90	42

optimized mutation is better than the random mutation at generating the inputs; those would not go to far positions from the target code areas. Precisely, the optimized mutation can achieve an average of 90% proportion in Det and 42% proportion in Havoc. On the contrary, the random mutation only obtains an average of 68% in Det and 25% in Havoc. Furthermore, Table 6 shows the evaluation results w.r.t. the AI2 appraisal index. On average, the optimized mutation is still shown to outperform the random mutation in generating the inputs with smaller input-distance; in other words, our optimized mutation manifests a stronger directed ability.

Besides, as can be seen, the values in Table 6 is smaller than those in Table 5; it is indicated that generating inputs with smaller input-distance is quite difficult; to some extent, this means that driving the program to the target code areas is not easy.

Overall, the analysis indicates that the answer to RQ1 is definite; the optimized mutation is able to improve the possibility of generating inputs closer to the target code areas.

4.6. Evaluation of the Directed Performance (RQ2). In this section, we are ready to make an investigation about the directed performance of RDFuzz and compare it with other directed/nondirected fuzzers.

To represent the directed performance quantitatively, we adopt the minimum input-distance among all the generated inputs as the appraisal index, which is a way to reflect the best result of the directed testing. Furthermore, we do a trick that if an input hits the target code areas and the program crashes at the same time, then the input-distance goes to 0 immediately.

Table 6: AI2: the proportion of the inputs with smaller input-distance among all the mutated inputs.

Benchmark	Random mutation		Optimized mutation	
	Det (%)	Havoc (%)	Det (%)	Havoc (%)
AFLGo-demo	13	2	22	2
CVE-2017-9050	10	5	25	6
CVE-2015-7497	15	2	19	7
CVE-2015-8241	10	1	15	5
CVE-2016-10094	11	7	12	10
CVE-2015-4695	12	7	22	7
CVE-2018-19519	19	5	29	11
Average	13	4	21	7

Moreover, the exploration result plays a significant role in the directed fuzzing, because an abundant exploration usually leads to a satisfactory exploitation result. Therefore, we also pay attention to the coverage result. As a choice, we take the branch coverage number as an indicator of the exploration results; in fact, it is the size of the covered bitmap, which is a fundamental mechanism provided by AFL.

In the performance comparison, it is necessary to take AFLGo into consideration. Since Hawkeye [2] is not yet available, we cannot run our own experiments comparisons against it. Besides, we also take two nondirected fuzzers into consideration, AFL and FairFuzz, because only the exploration process can sometimes cover the target code areas as well, so that taking some nondirected fuzzers into consideration can make the analysis more convincing.

Figure 8 shows the exploration and exploitation results on the 7 benchmarks. Based on these results, we gain the following observations.

(i) On the 7 benchmarks, reducing the minimum input-distance in the RDFuzz (the red line) shows a better drop than AFLGo (the green line). Besides, it can be seen that RDFuzz achieves better exploration results than AFLGo on 6 benchmarks out of the total 7 benchmarks. And on the last benchmark (CVE-2015-7497), the exploration result by RDFuzz is close to that by AFLGo.

In summary, it is suggested that RDFuzz is a better directed testing engine than AFLGo, at least on these 7 benchmarks. It is also proved that our improvements on exploration and exploitation, as well as the intertwined schedule, work well to provide a sufficient exploration result and obtain a good exploitation result.

(ii) On the benchmarks of CVE-2017-9050, CVE-2015-8241, and CVE-2018-19519, the exploitation results by FairFuzz are somewhat counterintuitive; i.e., it presents a better directed performance than the directed testing engine AFLGo. Besides, the final exploration results by FairFuzz are the best among the 4 testing engines.

The poor directed performance in AFLGo is due to the exploration and exploitation tradeoff problem; AFLGo can get stuck in the exploitation stage with an inadequate exploration result. On the contrary, FairFuzz is always in the exploration stage, aiming to improve the coverage, and it finds some inputs close to the target code areas, owing to its high exploration results.

This is just the reason why we still introduce the exploration strategy in RDFuzz. As can be seen, on these three benchmarks, the directed performance of RDFuzz is similar to FairFuzz and better than AFLGo. It is proved that RDFuzz can avoid being stuck by an inadequate exploration result.

Overall, the analysis indicates that the answer to RQ2 is definite; RDFuzz manifests a well directed performance in achieving the target code areas.

5. Threats to Validity

There are three threats to validity. The first threat is that the CG construction is limited by the identification of the indirect call, e.g., the function pointer [28], which can make the CG construction incomplete, and the graph-based distance calculation becomes inaccurate as a result.

The second threat is that it is still difficult for the directed testing to discover the code areas deeply buried in the program, whereas some improved exploration strategies are applied. This is due to the inherent limitation of the fuzzing technique, because it is clumsy at discovering the paths protected by the complex constraints.

The third threat is that our proposed approach is a heuristic solution to the balance problem between exploration and exploitation in the directed testing, yet the evaluation is not widely exampled. Because the real-world programs are variously complicated, we cannot ensure the effectiveness of our approach on any programs.

6. Future Work

As future work, we are planning to investigate some lightweight methods to make CG construction more complete, which is a significant step to provide accurate input evaluation. Moreover, we will explore some adaptive algorithms, which can bring better solutions to the tradeoff problem between exploration and exploitation. And we will make

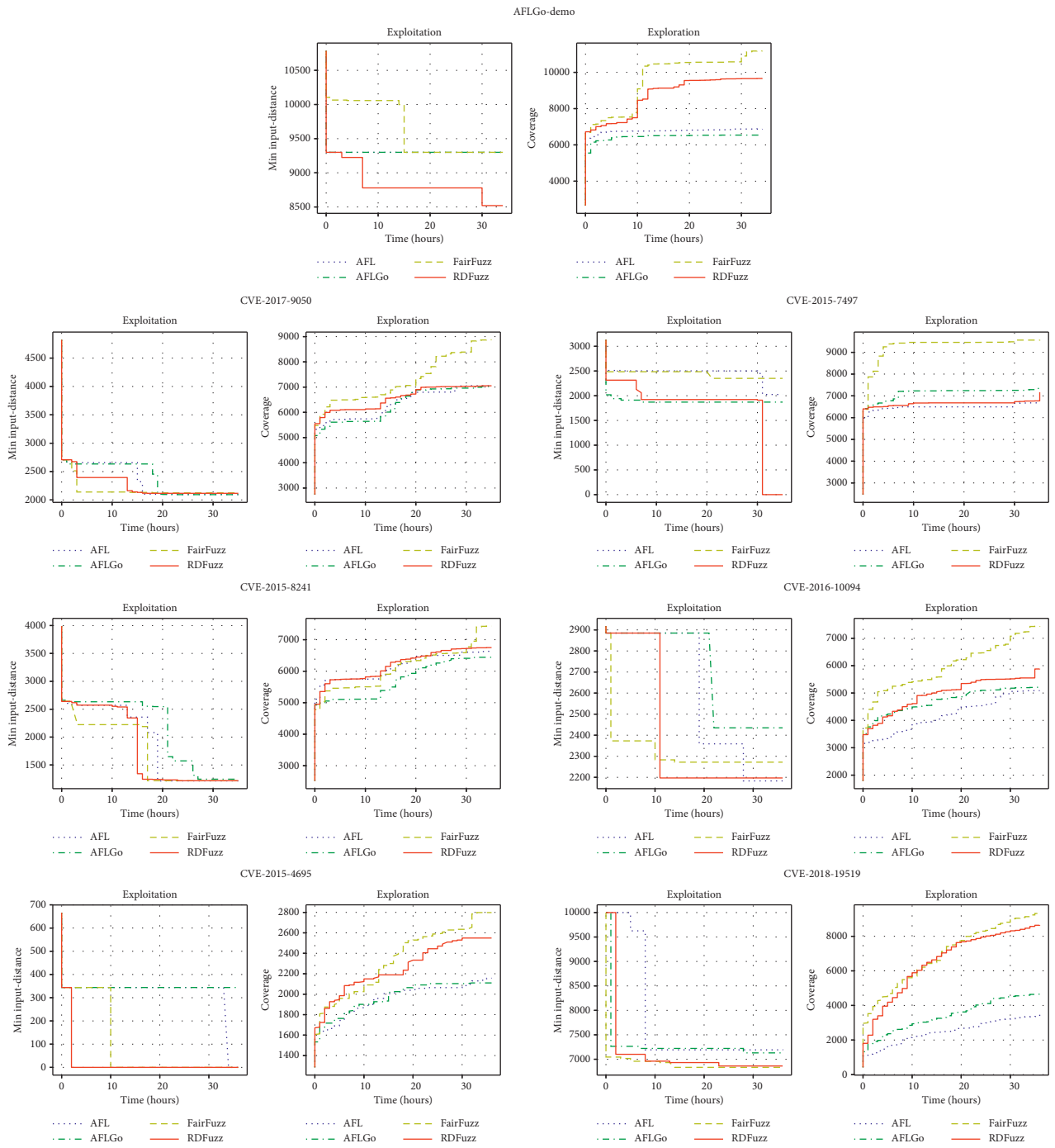


FIGURE 8: For each benchmark, the left subfigure shows the exploitation result, i.e., the minimum `input-distance` among all the generated inputs; the right subfigure shows the exploration result, i.e., the number of covered branches.

more evaluation on real-world programs to examine the effectiveness.

7. Conclusion

In this paper, we perform an investigation on the state-of-the-art directed fuzzing engine AFLGo and point out two main limitations. We argue that these two limitations are representative in the directed fuzzing scenarios. The first limitation is the balance problem between the exploration need and the exploitation need in the directed fuzzing. The second limitation is that the random mutation is blind and cannot steer the program toward the target code areas. We further propose a new tool, dubbed by `RDFuzz`, to provide a better directed performance. In `RDFuzz`, based on the input evaluation and classification, we apply an intertwined testing schedule and present some improvements on the exploration and exploitation stage, respectively. The

evaluation results demonstrate that RDFuzz is skilled at steering the program toward the target code areas and is not easy to get stuck into the local optimum.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (61602502).

References

[1] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pp. 2329–2344, ACM, New York, NY, USA, 2017.

[2] H. Chen, Y. Xue, Y. Li et al., "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pp. 2095–2108, ACM, Toronto, Canada, 2018.

[3] S. Godboley, A. Dutta, D. P. Mohapatra, and R. Mall, "Scaling modified condition/decision coverage using distributed concolic testing for Java programs," *Computer Standards & Interfaces*, vol. 59, pp. 61–86, 2018.

[4] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis*, E. Yahav, Ed., vol. 6887, pp. 95–111, Springer, Berlin, Germany, 2011.

[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 51–39, 2018.

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pp. 1032–1043, ACM, Vienna, Austria, 2016.

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[8] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[9] A. Rebert, S. K. Cha, T. Avgerinos et al., "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, USA, pp. 861–875, August 2014.

[10] V. J. M. Manes, H. S. Han, C. Han et al., "The art, science, and engineering of fuzzing: a survey," *IEEE Transactions on Software Engineering*, p. 1, 2019.

[11] S. Godboley, A. Dutta, D. Prasad Mohapatra, and R. Mall, "GECOJAP: a novel source-code preprocessing technique to improve code coverage," *Computer Standards & Interfaces*, vol. 55, pp. 27–46, 2018.

[12] C. Lemieux and K. 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, ASE 2018*, pp. 475–485, ACM, New York, NY, USA, 2018.

[13] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, San Francisco, CA, USA, May 2018.

[14] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: application-aware evolutionary fuzzing," in *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2017.

[15] S. Gan, C. Zhang, X. Qin et al., "CollAFL: path sensitive fuzzing," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, San Francisco, CA, USA, May 2018.

[16] D. Jaeger and R. Jung, Eds., *Encyclopedia of Computational Neuroscience*, p. 1649, Springer, New York, NY, USA, 2015.

[17] M. Zalewski, "Tool AFL mirrorer," 2017, https://github.com/mirrorer/afl.

[18] S. Pailoor, A. Aday, and S. Jana, "MoonShine: optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, Baltimore, MD, USA, pp. 729–743, August 2018.

[19] S. Godboley, S. Panda, A. Dutta, and D. P. Mohapatra, "An automated analysis of the branch coverage and energy consumption using concolic testing," *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 619–637, 2017.

[20] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pp. 75–86, IEEE, San Jose, CA, USA, 2004.

[21] S. Q. Liu and E. Kozan, "New graph-based algorithms to efficiently solve large scale open pit mining optimisation problems," *Expert Systems with Applications*, vol. 43, pp. 59–65, 2016.

[22] M. Zalewski, "American fuzzy lop (AFL)," 2015, http://lcamtuf.coredump.cx/afl/.

[23] Tool, AFLGo, 2019, https://github.com/aflgo/aflgo.

[24] OSS-Security, Invalid Writes and Reads in Libxml2, 2017, https://www.openwall.com/lists/oss-security/2017/05/15/1.

[25] CVE-Common Vulnerabilities and Exposures (CVE), https://cve.mitre.org/.

[26] K. Serebryany, D. Bruening, P. Alexander, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, USENIX., Boston, MA, USA, pp. 309–318, 2012.

[27] U. Kargén and N. Shahmehri, "Speeding up bug finding using focused fuzzing," in *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018*, 2018.

[28] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering—ICSE'18*, pp. 327–337, ACM Press, Gothenburg, Sweden, 2018.