

**ABSTRACT**

Directed grey-box fuzzing (DGF) is a security testing technique that aims to steer the fuzzer towards predefined target sites in the program. To gain directedness, DGF prioritizes the seeds whose execution traces are closer to the target sites. Therefore, evaluating the distance between the execution trace of a seed and the target sites (aka, the seed distance) is important for DGF. The first directed grey-box fuzzer, AFLGo, uses an approach of calculating the basic block level distances during static analysis and accumulating the distances of the executed basic blocks to compute the seed distance. Following AFLGo, most of the existing state-of-the-art DGF techniques use all the basic blocks on the execution trace and only the control flow information for seed distance calculation. However, not every basic block is equally important and there are certain basic blocks where the execution trace starts to deviate from the target sites (aka, deviation basic blocks).

In this paper, we propose a technique called WindRanger which leverages deviation basic blocks to facilitate DGF. To identify the deviation basic blocks, WindRanger applies both static reachability analysis and dynamic filtering. To conduct directed fuzzing, WindRanger uses the deviation basic blocks and their related data flow information for seed distance calculation, mutation, seed prioritization as well as explore-exploit scheduling. We evaluated WindRanger on 3 datasets consisting of 29 programs. The experimental results show that WindRanger outperforms AFLGo, AFL, and FairFuzz by reaching the target sites 21%, 34%, and 37% faster and detecting the target crashes 44%, 66%, and 77% faster respectively. Moreover, we found a 0-day vulnerability with a CVE ID assigned in ffmpeg (a popular multimedia library extensively fuzzed by OSS-fuzz) with WindRanger by supplying manually identified suspect locations as the target sites.

---

**KEYWORDS**

Fuzz Testing, Directed Testing

---

**CCS CONCEPTS**

- Software and application security; Software security engineering; Security and privacy

---

**1 INTRODUCTION**

Software vulnerability is a major threat to software security. Fuzzing is a software testing technique for finding vulnerabilities. The key idea of fuzzing is to feed the program under test (PUT) with a large amount of test inputs and observe for abnormal behaviors (e.g., crashes) of the program. According to how much information about the program internals is used, fuzzing techniques can be categorized as blackbox, greybox and whitebox. Blackbox fuzzing techniques require no knowledge about the internals of the PUT and they may lack effectiveness when testing programs with complex states. Whitebox fuzzing techniques captures the entire execution context (memory, registers etc.) of the PUT and they often apply techniques such as symbolic execution or taint analysis which may limit their applicability on large programs. Greybox fuzzing techniques use light-weight program instrumentations to collect the run-time information needed for guiding the fuzzing process. Greybox fuzzing has the best of both worlds since it is both effective enough to detect deeply embedded bugs and scalable enough to be applied to large programs. Therefore, greybox fuzzing is widely used by both practitioners and researchers [25, 30, 34, 40, 43, 45].

Despite being highly effective, greybox fuzzing generally lacks the directedness to reach predefined target locations in the PUT (aka, target sites 1). To address this issue, Böhme et al. proposed the concept of Directed Greybox Fuzzing (DGF) [3] in 2017. The key feature of DGF is that it spends most of its time budget on reaching and testing the target sites without wasting resources
AFLGo via static analyses can be set as the target sites and with the DGF the inputs preserved for subsequent fuzzing as seeds. To find solutions for the aforementioned pitfalls, we scrutinize the execution traces failing to reach the target sites and observe that for every execution trace, there are some key basic blocks where it starts to deviate from the target sites.

The main contributions of this paper are summarized as follows:

- We present the concept of deviation basic block and an approach of identifying the deviation basic blocks with static and dynamic analysis.
- We present WindRanger, a directed grey-box fuzzer leverages the deviation basic block information to augment seed scheduling and mutation strategies.
- We implement the prototype for WindRanger and publish the source code for future research.
- We evaluate WindRanger thoroughly with extensive experiments and demonstrate the superiority of WindRanger.

In this paper, we denote all the files fed to the program in fuzzing as inputs, and only the inputs preserved for subsequent fuzzing as seeds.
This paper is accompanied by a website: https://sites.google.com/view/windranger-directed-fuzzing/. The experimental data and source code are available on this website.

2 MOTIVATING EXAMPLE

Figure 1 shows a code snippet crafted from CVE-2018-8962 [32], where Use-After-Free bugs can be triggered when the program calls the getName function from the decompileJUMP function. Besides the source code, Figure 1 also shows the corresponding control flow graph of the code snippet where each node represents a basic block.

Suppose we are trying to use DGF to reproduce this CVE, we will need to set the basic blocks containing line 16 and line 22 as the target sites and calculate the distance of each basic block to the target sites accordingly. For simplicity, here we use the number of edges between a basic block and the nearest target site as its distance value. In the control flow graph in Figure 1, the notation of X.Y means that X is the line number of the first instruction of the basic block and Y is the distance of the basic block towards the target. If the target sites are not reachable from a basic block, only the line number is labeled. For example, node 6:3 means the basic block starts from line six and it can reach the nearest target site on line 16 by traversing three edges. Another example is node 18, which starts from line 18 that is a return. Since the program can never reach the target sites after the return, no distance is assigned to node 18.

Limitations of Existing Approaches. Assume the directed greybox fuzzer has found three seeds (A, B and C), and their execution traces are shown in Figure 1. For the execution traces of seeds in Figure 1, the executed basic blocks are marked with dark background and the executed edges are represented with dashed lines. Now, we need to calculate the distance of each seed to decide which one should be prioritized. Following the idea of [3, 8, 50], where the distance of a seed is the arithmetic mean of the distances of all the basic blocks on its execution trace, the distances of seed A, B and C are \( \frac{1+3+1}{3} = 2.33 \), \( \frac{1+4+2}{3} = 2.33 \), and \( \frac{2+2+3}{3} = 2 \) respectively. Based on these distances, it seems seed C should be preferred. However, in reality, the condition on line 15 is very hard for the fuzzer to penetrate through, meaning that the edge between node 15:1 and node 16 can hardly be executed. As a result, seed C is not a good choice if we consider the path conditions. The discussion about seed C shows that \textit{only using the control flow information to calculate the distance is not enough, data flow information should also be considered.} Now that seed C has been excluded, we need to select from seed A and seed B. At first glance, seed B has a smaller distance (2.33 vs 2.86) and the fuzzer should prefer seed B. Nonetheless, if we check the execution trace, we can see that the execution trace of seed A is actually closer to the target site on line 22 and seed A is the better one to select. The reason for seed A to have a larger distance is that it executes node 7:5, node 9:4 and node 12:3, which increases the overall seed distance. The comparison between seed A and B shows that \textit{the seed distance calculated with all the basic blocks can be biased and thus inaccurate.}

Our Observations. From the execution traces in Figure 1, we can see that for an execution trace that fails to reach the target sites, it must deviate from the target sites at certain basic blocks. For example, for the execution trace of seed A, when it reaches node 21:1, it deviates from the target by not entering the if condition. Note that node 14:2 is not a deviation point but both edges extended from this node can possibly reach a target site. Similarly, for seed B, the deviation point is node 20:2 and for seed C, the deviation point is node 15:1. In this paper, we call such basic blocks the \textit{deviation basic blocks (DBBs)}.

Our Approach. If we calculate the distances by averaging only the DBBs, the seed distances for A, B and C are 1, 2 and 1 respectively. If we can further adjust the distance of seed C according to the hard-to-meet condition on line 15, we can eventually prioritize seed A, the actual best seed here. As we have demonstrated the importance of the DBBs, the problem now becomes how to identify them.

Although in this example, all the DBBs happen to be the basic blocks with the shortest distance, the identification of them is not that straightforward. Suppose we have another seed D, whose execution trace is \(<6:3 \rightarrow 7:5 \rightarrow 9:4 \rightarrow 10\rangle\). For this seed, even though node 6:3 has a shorter distance than node 9:4, the latter one should be the DBB since the execution trace actually deviates from the target sites at node 9:4. This is because the successor of node 9:4 is node 10, which is unreachable to the target sites, not node 12:3 which is reachable. Altering the branching on node 9:4 can affect whether the target sites can be reached or not. On the contrary, no matter how we alter the branching at node 6:3, the execution trace always has a chance to reach at least one target site. In WindRanger, we give a clear definition of DBBs in the context of DGF and propose an approach to identify them on execution traces. Furthermore, we propose an approach to leverage the DBBs information as well as data flow information for better seed distance calculation. Last but not least, we integrate the improvements into different components of DGF to maximize the performance boost.

3 APPROACH OF WINDRANGER

Figure 2 shows the overview of WindRanger. WindRanger contains two major components: static analysis and fuzzing. The static analysis of WindRanger is used to identify the plausible candidates of the deviation basic blocks (DBBs), calculate their distance, and instrument this information into the target program. Moreover, like in CGF, the instrumentation can also collect edge coverage to guide the grey-box exploration of different program states.

In the fuzzing process, before selecting a seed input from the seed queue, WindRanger will first decide whether it should run in the exploration mode or the exploitation mode (①). The switching is decided dynamically to balance between exploration and exploitation and avoid getting stuck in local optimums. In the exploration stage, WindRanger works like a normal coverage-guided grey-box fuzzer. In the exploitation stage, WindRanger prioritizes the seeds with shorter distances via selecting them first and allocating more energy for mutating them (②). After a seed is selected from the queue and allocated a power schedule (to determine how many new test inputs to be generated from it), WindRanger will conduct mutation (③) augmented with the results of the data flow analysis.
(5). The mutation helps to steer the fuzzer to pass the conditionals of the DBBs. Then, after a mutated input is executed, WindRanger will analyze the execution traces to decide whether to keep the input as a new seed and if the input is kept as a new seed, WindRanger will also confirm the corresponding DBBs (5). Before adding the seed to the seed queue, WindRanger will calculate its distance towards the target sites according to the DBBs (§3.1) and adjust the distance value according to the data flow analysis results (5). This marks the end of one iteration in the fuzzing loop.

In the following sections, we will explain in detail the following questions: ① how does WindRanger identify the DBBs for a seed using static and dynamic analyses (§3.1); ② how does WindRanger perform the probing based taint analysis (§3.2); ③ how does WindRanger calculate the seed distance and fine-tune it with the taint analysis result (§3.3); ④ how does WindRanger mutate the input with the taint analysis result (§3.4); ⑤ how does WindRanger make decisions on switching between exploration and exploitation (§3.6).

### 3.1 Deviation Basic Blocks Identification

Deviation basic block (DBB) is an important concept in WindRanger. To identify the DBBs in the execution trace of a seed, WindRanger first finds the potential DBBs in the program with static analysis. Then, during fuzzing, WindRanger pinpoints the DBBs in the execution trace with analyses on executed basic blocks and their relations with the potential DBBs.

#### Static DBB Collection

Here we provide the definition of the potential DBBs for a program and then discuss how WindRanger finds the potential DBBs with static analysis.

**Definition 1 (Potential DBBs for a program).** Given a program $P$, all its basic blocks $AIBB(P)$ and a set of target sites $T$, the potential DBBs of $P$ (denoted as $\Phi(P)$) is a set of basic blocks:

$$\Phi(P) = \{ b \in AIBB(P) \mid isReachable(b, T) \land (\exists c \in Successors(b)) [\neg isReachable(c, T)] \}$$  

(1)

where $isReachable(x, T)$ is a function which returns true if the basic block $x$ is reachable to at least one target site in $T$ on the interprocedural control flow graph and returns false otherwise; $Successors(b)$ is the set of all succeeding basic blocks of $b$.

In plain text, Definition 1 can be explained as: the potential DBBs of a program are the basic blocks which can reach to at least one target and at least one of its succeeding basic blocks is not reachable to any target. The rationale of this definition is double-fold. First, a potential DBB must be reachable to the target sites, otherwise, it cannot help to reach the target sites, and there is no point to analyze...
the deviation issue on the non-reachable basic blocks. Second, at least one of the successors of a potential DBB is not reachable, because, if all successors of a basic block are reachable to the target, execution traces cannot deviate from the targets on this basic block.

In practice, WindRanger constructs an interprocedural Control Flow Graph (iCFG) of the target program and computes the reachability for each basic block and the target sites base on the iCFG. A basic block is considered reachable to a target site if there exists a path from the basic block to the target site on the iCFG. Then for each basic block that is reachable to target sites, if at least one of its successor is unreachable to any target, WindRanger adds this basic block to the \( \Phi(P) \).

**Dynamic DBB Collection.** Here we define the DBBs of an execution trace and then discuss how WindRanger identifies the DBBs dynamically.

**Definition 2** (DBBs for a seed). Given the potential DBBs \( \Phi(P) \) and a seed \( s \), the DBBs of the execution trace of \( s \) (denoted as \( \Phi(s) \)) is a set of basic blocks:

\[
\Phi(s) = \{ b \in \xi(s) | b \in \Phi(P) \land (\forall c \in \text{ReachableSucc}(b))[c \notin \xi(s)] \}
\]

where \( \xi(s) \) denotes all basic blocks in the execution trace of the seed \( s \); \( \text{ReachableSucc}(b) \) is all successors of the basic block \( b \) which can reach to the target sites.

In plain text, Definition 2 can be explained as: the DBBs of a seed are the basic blocks in its execution traces which are the potential DBBs of the program and all of their reachable successors are not executed by this seed. The rationale of this definition is double-fold. First, a DBB of the execution trace, by nature, must be one of the potential DBBs of the program. Second, all of the reachable successors of a DBB should not appear in the execution trace. The reason for the second point is that if one of the reachable successors of a basic block is in its execution trace, the program is running towards (instead of away from) the targets on this basic block and thus the basic block is not an actual DBB for this execution trace.

The potential DBBs for a program are used as intermediate results to acquire the DBBs for seeds that are used for guiding the DGF. Thus, in the following sections discussing the key steps of the fuzzing process, if not specially mentioned, the term DBB refers to DBB for seeds.

### 3.2 Probing-based Taint Analysis

WindRanger uses probing-based taint analysis similar to FairFuzz [25] and GreyOne [16] to collect data flow information about which bytes in the seed can influence the constraints for a given branch. The data flow information is then stored in a hash map called effector map, where the key is the basic block address for a branch constraint and the value is a set of indexes of the bytes on the seed which can influence the branch constraint. The effector map is used for seed distance calculation (§ 3.3) and data sensitive mutation (§ 3.4).

Algorithm 1 shows the process of generating the effector map, where \( \text{ValueOf}(\text{var}, s) \) returns the value of variable \( \text{var} \) for a seed \( s \). For a seed \( s \) and its execution trace \( \xi(s) \), WindRanger first extracts all the variables related to branch constraints on the execution trace (\( \forall \text{VAR} \)). Then, WindRanger mutates the seed byte by byte with a set of predefined mutation operators (\( \text{OP} \)). With each mutated input, WindRanger checks for every extracted variable whether its value remains unchanged after the mutation. If the variable value is changed, WindRanger updates the effector map to take note that the mutated position of seed \( s \) can affect the variable.

In the current implementation of WindRanger, we predetermine three mutation operations for \( \text{OP} \): 1) Bit-flip. Modify the byte at the given position by flipping each bit of it. 2) Insertion. Insert a random byte at the given position. 3) Deletion. Delete the byte at the given position.

### 3.3 Seed Distance Calculation

Seed distance calculation is important in DGF. Seeds with shorter distances are prioritized and allocated with more energy for mutation.

With the target basic blocks \( T_B \), the distance of a seed \( s \) is:

\[
d_s(s, T_B) = \frac{\sum_{m \in \Phi(s)} d_s(m, T_B)}{|\Phi(s)|}
\]

where \( d_s(m, T_B) \) is the distance between DBB \( m \) and the target basic blocks; \( \Phi(s) \) consist of the DBBs in the execution trace of seed \( s \). In other words, for seed \( s \), Equation 3 uses the average distance of the DBBs in its execution trace as its distance to target basic blocks \( T_B \).

Note that structure of Equation 3 and the definition of \( d_s(m, T_B) \) are both the same as in AFLGo [3] and HawkEYE [8] except that we use the DBBs instead of all the basic blocks to calculate the distance.

However, Equation 3 is still not accurate enough since only control flow information is used for the distance calculation. We need to further balance the distance of each DBB with the data flow information and the equation now becomes:

\[
d_s(s, T_B) = \frac{\sum_{m \in \Phi(s)} d_s(m, T_B) \cdot \Psi(s, m)}{|\Phi(s)|}
\]

where \( \Psi(s, m) \) is used to judge the degree of difficulty to penetrate the constraint on DBB \( m \) with seed \( s \). \( \Psi(s, m) \) is defined as:

\[
\Psi(s, m) = \min(\text{Max}_{\gamma} \left\lfloor \frac{\text{NumOfEffectiveBytes}(s, m)}{\gamma} \right\rfloor)
\]

where \( \text{NumOfEffectiveBytes}(s, m) \) indicates the number of input bytes that can affect the constraint variable(s) of \( m \) according to
the effector map of \( s \); \( y \) is a constant ratio which controls the granularity; \( \text{Max}_y \) is a constant which is the upper bound of \( \Psi(s, m) \). The intuition here is that the more input bytes are influencing the constraint, the more difficult it is for the fuzzer to penetrate the constraint.

We use the number of bytes affecting the constraints instead of branch distances mainly because branch distances cannot reflect the difficulty of penetrating a branch if the input content goes through complex data transformations before reaching the branch conditions. For example, suppose we have an if condition comparing the md5sum of the input content with a constant string. Assume, the constant string is DEADBEF and the md5sum of the input is HEADBEF. If we use branch distance here, the calculated distance is very short since the md5sum is only one byte away from matching the constant string. However, the fuzzer will need to modify the content of the entire input to make the md5sum match the constant string. Therefore, in this case, using the number of bytes affecting the constraints is a better choice. In addition, we empirically found in the experiments that using the number of bytes affecting the constraints can yield good results (§ 4).

3.4 Data Flow Sensitive Mutation

With the information provided by the effector map, WindRanger adopts data flow sensitive mutation to better penetrate branch conditions. Given a seed and its effector map, WindRanger first checks if it is running in the exploitation stage or exploration stage. If WindRanger is running in the exploitation stage, it identifies the input bytes related to the constraint variables of the DBBs as high priority bytes. Otherwise, it identifies the input byte related to all the constraint variables as high priority bytes. During the random mutation, high priority bytes are given a larger chance of mutation.

In addition, for a constraint variable and its related input bytes, if the related input bytes are consecutive, WindRanger also checks if the variable and the related input bytes share the same value. If they have the same value, it is likely that the input bytes did not go through data transformations. In this case, WindRanger will try to replace the related input bytes with the value of the other comparison operand of the constraint before applying random mutations.

3.5 Seed Prioritization

Similar to data flow sensitive mutation, the seed prioritization also works differently during the exploitation stage and exploration stage. During the exploitation stage, WindRanger maintains a two-level priority queue. First, for each DBB, WindRanger finds a list of seeds that can cover this DBB. Then, WindRanger sorts the list of every DBB in ascending order according to the distances of the seeds. Lastly, WindRanger selects the first seed on every list and puts them into the favored queue with higher priority, and puts the rest of the seeds into the less favored queue. When selecting the next seed for mutation, WindRanger has a higher chance to select from the favored queue. During the exploration stage, WindRanger works like a normal CGF and prioritizes the seeds which can help to achieve better code coverage.

As for power scheduling, WindRanger follows the simulated annealing algorithm proposed in AFLGo [3]. The only difference is the seed distance used for energy calculation.

3.6 Dynamic Switch between Explore and Exploit Stage

Although the purpose of DGF is to reach the targets as fast as possible, DGF still needs enough coverage exploration to avoid getting stuck in local optima. For example, AFLGo adopts a user-defined time budget in advance to split the exploration stage and exploitation stage. However, strong domain knowledge is required for the users to find suitable time budgets for different programs. To address this issue, we propose to dynamically switch between the exploration and exploitation stage according to the execution status of DBBs.

Specifically, during fuzzing, WindRanger maintains all existing DBBs as a global set (denoted as \( \Phi(F) \)). In the exploit stage, when all DBBs in \( \Phi(F) \) have been exploited adequately, WindRanger switches to the exploration stage. Whether a DBB has been exploited adequately is decided according to how many times the DBB has been executed.

Formally, whether an input \( x \) can hit a basic block \( b \) (denoted as \( \text{hit}(x, b) \)) or not can be defined as

\[
\text{hit}(x, b) = \begin{cases} 
1, & b \in \xi(x) \\
0, & b \notin \xi(x) 
\end{cases}
\]

where \( \xi(x) \) consist of all basic blocks in the execution trace of \( x \).

During fuzzing, WindRanger records the number of how many times each basic block has been hit. Formally, the hit count of a basic block \( b \) is

\[
\text{numHit}(b) = \sum_{x \in \Gamma} \text{hit}(x, b)
\]

where \( \Gamma \) is the set of all inputs produced by fuzzing so far.

Based on the hit count, a basic block \( b \) is considered as exploited adequately such that

\[
\text{numHit}(b) > v \cdot \min_{b' \in T} \text{numHit}(b')
\]

where \( T \) is the set of all basic blocks WindRanger has encountered during fuzzing; \( v \) is a constant factor.

As for switching from the exploration stage to the exploitation stage, if WindRanger discovers new DBB during the exploration stage, it will switch to the exploitation stage. Note that the switching between stages always happens after WindRanger finishes mutating a seed and before it selects a new seed to mutate, as shown in Figure 2.

4 IMPLEMENTATION & EVALUATION

The implementation of WindRanger mainly consists of two parts: static analyzer and dynamic fuzzer. For the static analyzer, we leverage the static analysis framework SVF [38] to construct iCFG from LLVM IR. The static analyzer is implemented with about 900 lines of C/C++ code. The dynamic fuzzer is implemented based on AFL [23], version 2.52b, with about 1000 lines of C code.

With the implemented prototype of WindRanger, we conducted large scale experiments to answer the following research questions:
RQ1: How good is the ability of WindRanger for reaching predefined target sites?
RQ2: How good is the performance of WindRanger in terms of reproducing the target bugs?
RQ3: How does every component in WindRanger affect the overall performance?
RQ4: Can WindRanger help in real-world bug hunting?

4.1 Evaluation Setup

Evaluation Datasets. We use programs from the following datasets as the evaluation benchmarks:

- UniBench [27] is a recent dataset proposed to evaluate fuzzing works. It contains 20 real-world programs of 6 different categories (categorized according to the input file type). We selected 4 target sites per program from this dataset to evaluate the Time-to-Targets (TTT) of different techniques. This set of benchmarks are used to answer RQ1 and RQ3.
- AFLGo Test Suite [3] is a set of programs with n-day vulnerabilities used in the experiments of AFLGo [5]. This test suite has been used in several research works [3, 8] to evaluate DGF techniques. This set of benchmarks are used to answer RQ2.
- Fuzzer Test Suite [18] is a set of fuzzing benchmarks proposed by Google. It contains several real-world programs with known vulnerability information and is a common benchmark used in fuzzing research [10, 21]. This set of benchmarks are used to answer RQ2.

Evaluated Techniques. Since WindRanger is built on the top of AFL, we compare WindRanger with several AFL-based fuzzers to avoid the potential bias caused by the implementation of the techniques:

- WindRanger is the tool proposed in this paper. We empirically set the values of various configurable options (such as γ in Equation 5). All the settings for the configurable options are available on our website [14].
- AFLGo [3] is a state-of-the-art DGF technique. Other directed grey-box fuzzers (e.g. HawkEye [8], FuzzGuard [49], etc.) are not publicly available by the time of writing this paper.
- AFL [23] is a representative coverage-guided grey-box fuzzer which is the basis of many other fuzzing techniques [3, 16, 25, 47].
- FairFuzz [25] is a derivation of AFL. We chose FairFuzz for comparison because FairFuzz also uses the probing technique to collect taint information to facilitate fuzzing. Note that although FairFuzz shares similarities with WindRanger, it still represents advanced CGF techniques since its purpose is to maximize code coverage.

By comparing with the CGF techniques, we can demonstrate that WindRanger can truly bring directedness to grey-box fuzzers. By comparing with AFLGo, we can demonstrate the effectiveness of the strategies proposed in WindRanger.

Evaluation Criteria. We use two types of criteria to evaluate the performance of different techniques:

- Time-to-Target (TTT) is used to measure the time used by the fuzzer to generate the first input which can hit the specified target site. This criterion is used to evaluate the techniques on benchmarks without known bugs.
- Time-to-Exposure (TTE) is used to measure the time used by the fuzzer to trigger a known bug. This criterion is used to evaluate the techniques on benchmarks with known bugs.

Experiments Settings. By default, all the experiments were repeated 10 times with time budgets of 24 hours, except for the experiments on the AFLGo Test Suite. We followed the setup in the paper of AFLGo where the experiments were repeated 20 times with time budgets of 8 hours. For the statistical test, we use the Mann-Whitney U test (p-value) to measure the statistical significance of the experiment results. Moreover, we also use the Vargha-Delaney statistic (A12) [39] to measure the chance for one technique to perform better than another.

Experiments Environment. We conducted the experiments on machines equipped with Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz with 48 cores, using Ubuntu 20.04.3 LTS as the operating system. During experiments, each fuzzer instance runs in a docker container and binds to one CPU core.

4.2 Target Site Reaching Capability (RQ1)

Currently, there is no standard dataset for evaluating TTT for DGF techniques. Although Google fuzzer test suite contains three projects (libjpeg-turbo-07-2017, libpng 1.2.56 and freetype2-2017) with given target sites to test the ability of fuzzers to cover these locations, we found them insufficient. First, the amount of target sites is small (only 5). Second, using AFL-based fuzzers, some target sites are easy to reach (1 second) with provided seeds or can not reach within the time budget (24 hours) without provided seeds because this test suite was initially designed for LibFuzzer.

To get a dataset with abundant and valid target sites, we use the UniBench dataset which has diverse programs. To collect the target sites from the programs, we use the following steps: For each program, first we run AFL for 24h, repeat 10 times. During each run, we record the first-reach time (the time cost to generate the first seed to cover the basic block) for each basic block; Then, we calculate the average first-reach time across the 10 runs for each basic block; Based on the average first-reach time for each basic block, we use four scale time values (5 hours, 10 hours, 15 hours, 20 hours) to select four target sites. Normally, due to the page limitation, we conduct the experiments on all the programs in UniBench (the results for all the programs are on our website [14]). However, due to the page limitation, we select 6 programs (one program for a program type) and show the results in Table 1. On most of the target sites (20/24), WindRanger outperforms all other fuzzers and achieves the shortest μTTT. Overall, in term of mean TTT, WindRanger outperforms CGFs (AFLFairFuzz) by 56% and 55% respectively and outperforms DGF (AFLGo) by 31%. Besides, WindRanger outperforms AFLGo, AFL and FairFuzz by 21%, 34% and 37% respectively on all the programs. According to the values of mean A12 against other fuzzers (0.75 with AFL and Fairfuzz, 0.67 with AFLGo), we have enough...
Table 1: Time-to-target results on programs from UniBench. For each target site, the statistically significant (p-value < 0.05) values of \( \hat{A}_{12} \) is highlighted in bold; the shortest mean TTT is marked with an asterisk.

<table>
<thead>
<tr>
<th>Prog</th>
<th>Targets</th>
<th>AQL</th>
<th>AQLGO</th>
<th>FairFuzz</th>
<th>WindRanger</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runs ( \mu )T</td>
<td>Factor ( \hat{A}_{12} )</td>
<td>Runs ( \mu )T</td>
<td>Factor ( \hat{A}_{12} )</td>
<td>Runs ( \mu )T</td>
</tr>
<tr>
<td>jpe_cs.c:316</td>
<td>10 1h16m 1.43</td>
<td>0.87</td>
<td>10 1h15m 1.68</td>
<td>0.82</td>
<td>10 1h36m 2.14</td>
</tr>
<tr>
<td>libmp.dec:271</td>
<td>10 2h12m 1.70</td>
<td>0.86</td>
<td>10 1h40m 1.61</td>
<td>0.75</td>
<td>10 2h12m 2.11</td>
</tr>
<tr>
<td>jas_image.c:378</td>
<td>10 5h39m 1.26</td>
<td>0.70</td>
<td>10 7h42m 1.50</td>
<td>0.80</td>
<td>10 5h51m 1.14</td>
</tr>
<tr>
<td>jas_image.h:1026</td>
<td>7 1h15m 1.42</td>
<td>0.82</td>
<td>8 15h24m 1.35</td>
<td>0.73</td>
<td>7 1h60m 1.54</td>
</tr>
<tr>
<td>gain_analys.c:224</td>
<td>10 1h10m 3.31</td>
<td>0.90</td>
<td>10 51m33s 2.30</td>
<td>0.88</td>
<td>10 1h17m 3.40</td>
</tr>
<tr>
<td>bitstream.c:399</td>
<td>10 1h16m 1.27</td>
<td>0.69</td>
<td>10 3h59m 0.96</td>
<td>0.47</td>
<td>10 5h39m 1.40</td>
</tr>
<tr>
<td>get_audio.c:1452</td>
<td>10 1h16m 1.27</td>
<td>0.69</td>
<td>10 3h59m 0.96</td>
<td>0.47</td>
<td>10 5h39m 1.40</td>
</tr>
<tr>
<td>mpilib_interfce.c:142</td>
<td>7 1h15m 1.64</td>
<td>0.77</td>
<td>8 15h33m 1.43</td>
<td>0.64</td>
<td>8 18h13m 1.52</td>
</tr>
<tr>
<td>mpios2.a</td>
<td>10 5h39m 1.26</td>
<td>0.73</td>
<td>10 6h48m 1.26</td>
<td>0.73</td>
<td>10 1h17m 3.40</td>
</tr>
<tr>
<td>mpilib_interfce.c:142</td>
<td>7 1h15m 1.64</td>
<td>0.77</td>
<td>8 15h33m 1.43</td>
<td>0.64</td>
<td>8 18h13m 1.52</td>
</tr>
<tr>
<td>mpios2.a</td>
<td>10 5h39m 1.26</td>
<td>0.73</td>
<td>10 6h48m 1.26</td>
<td>0.73</td>
<td>10 1h17m 3.40</td>
</tr>
<tr>
<td>mpios2.a</td>
<td>10 5h39m 1.26</td>
<td>0.73</td>
<td>10 6h48m 1.26</td>
<td>0.73</td>
<td>10 1h17m 3.40</td>
</tr>
</tbody>
</table>

4.3 Bug Reproducing Capability (RQ2)

Reproducing bugs/crashes is an important application of directed fuzzing. Here we study the bug reproducing capability of WindRanger by comparing it with the baselines using the TTE criterion.

4.3.1 AFLGo Test Suite. In the paper [3] and website [5] of AFLGo, there are some programs with known vulnerabilities be used to evaluate the directness of DGF. Hence, we compare the performance of WindRanger on these programs directly with other tools. Specially, GUN Binutils 2.26 [2] and Libming 0.4.8 [29] with 9 CVEs are used in this experiment. We follow the same experiment settings in [3]; each instance conducted 20 times; with the time budget set to 8 hours; use an empty file as initial seed.

This test suite consists of programs from GUN Binutils 2.26 [2] and Libming 0.4.8 [29] with 9 CVEs. However, in the preliminary experiments, we found that plenty of crashes arise when fuzzing Binutils 2.26 but only 7 CVEs among them were used in the experiments of AFLGo. It requires cumbersome manual efforts to classify these crashes to find the corresponding CVEs. To minimize the deviation caused by manual labeling and improve the soundness of the results, we conduct experiments on Binutil 2.28 in which all the CVEs used in the experiment have been patched. For each CVE, we reverse its related patch(es) to ensure only this CVE is exposed in the experiments.

Table 2 shows the results from which we can observe that: 1) For CVEs which are difficult to expose (\( \mu \)TTE > 1 hour), WindRanger significantly outperforms other tools by 1.22x to 2.2x faster to expose them. For CWE-2016-4491 and CWE-2016-6131, WindRanger get the most hitting rounds (6 and 8 rounds). 2) The value of \( \hat{A}_{12} \) shows that WindRanger steadily achieves better performance, especially for CWE-2018-8807 and CWE-2018-8962, of which \( \hat{A}_{12} \) is equal or greater than 0.75 against other tools. 3) For CWE-2016-4489 and CWE-2016-4490, WindRanger do not exhibit better performance than AFLGo. However, these CVEs are easy to expose which only need few minutes. On these CVEs, the randomness of fuzzing matters for \( \mu \)TTE.

4.3.2 Fuzzer Test Suite. Compared to the bugs in AFLGo test suite, the bugs in Google fuzzer test suite require no classification since all bugs/crashes are easy to reproduce. Therefore, we conduct experiments on 13 bugs/crashes (\( \mu \)TTE > 1 hour) with 9 CVEs. We again use the short test suites in [3] and conduct each instance for 20 rounds; with the time budget set to 8 hours; use an empty file as initial seed.

The experiment results are shown in Table 3. The results in Table 3 demonstrate that by setting vulnerabilities locations as target sites, DGFs (WindRanger and AFLGo) can achieve better performance on exposing relative crashes than CFGs (AFL and FAIRFuzz). Among the 11 vulnerabilities, WindRanger gains better \( \mu \)TTE on 9. Overall, WindRanger achieves a mean speed up in TTE of 47% against AFL and 59% against FAIRFuzz. Compared to AFLGo, the 30% speed up shows that WindRanger owns better directedness capability. Moreover, the 30% speed up shows that WindRanger owns better directedness capability. Moreover,
### 4.4 Impact of Different Components (RQ3)

To investigate the impact of different components in WindRanger, we disable each component individually and conduct experiments on the same targets selected from UniBench as in § 4.2.

Table 4 shows the results. We can observe that disabling each component causes an increase (\(>10\%\)) on TTT, which means each component has a significant impact on the performance of WindRanger. Among the four components, the data flow sensitive mutation component contributes the most with a 22% speed up on TTE. By disabling other components (distance calculation, seed prioritization, and explore-exploit stage switch), we also see an increase in TTE of 12%, 17%, and 15% respectively. Note that all these components are related to the concept of DBB, which demonstrates the usefulness of DBB in DGF.

![Figure 3: The time cost to expose the new vulnerability in 10 runs](image)

**Figure 3**: The time cost to expose the new vulnerability in 10 runs

To check the if WindRanger can be used for hunting previously unknown bugs, we gave the prototype of WindRanger to a security expert. This security expert tried WindRanger on an interesting target, ffmpeg [15], which is a popular multimedia library and is well-fuzzed [17]. It is heavily fuzzed by OSS-fuzz with clusters of machines and using conventional fuzzing strategies can hardly yield any crash/bug. Therefore, the security expert conducted manual code review to identify suspicious locations before fuzzing. To be specific, she selected 50 assertions as the interesting locations to test (Details about the assertion locations can be found on our web site [14]).

Then, she tried to build a good enough seed (though cannot reach any of the targets) and fed the interesting locations as target sites to WindRanger to perform directed fuzzing. Eventually, she found one assertion fail among 50 assertions on the development branch. Note that on the release branch, all assertions will be removed and this assertion fail can possibly lead to security issues. The assertion fail has been confirmed as a vulnerability and fixed by the developers. Moreover, it received a CVE-ID (CVE-2021-38291).

Then we took the initial seed together with the vulnerable version of ffmpeg and used the vulnerability as the target site, to compare WindRanger with AFLGo on exposing the new vulnerability. Figure 3 shows the results. The horizontal axis indicates the time cost to expose (TTTE) the vulnerability. The boxes in the graph indicate the TTE results of corresponding tools in 10 runs. From Figure 3, we can observe that WindRanger costs less time to expose the vulnerability in almost every run. Specifically, the median TTE of WindRanger (8220s) is 1.64\times smaller than the median TTE of AFLGo (13440s).

### 5 THREATS TO VALIDITY

The first internal threat comes from the accuracy of the reachability analysis, which directly affects the identification of DBBs. The current implementation of WindRanger uses the SVF framework [38] to obtain the iCFG to analyze the reachability of the basic blocks. The iCFG can be erroneous due to some pitfalls such as inaccurate

\[\text{three vulnerabilities only take a few minutes or seconds to trigger (note that the two vulnerabilities on which WindRanger does not achieve the best results both belong to these three vulnerabilities). Normally these easy-to-trigger crashes can hardly benefit from DGF, and if we only consider the vulnerabilities whose \(\mu\)-TTTE exceed 1 hour, WindRanger can achieve larger performance gain with speed up of 66% against AFL, 44% against AFLGo and 77% against FairFuzz.}\]
Table 3: Time for exposing the known crashes in Google fuzzer test suite. For each vulnerability, the statistically significant ($p$-value < 0.05) values of $A_{12}$ is highlighted in bold; the shortest mean TTE is marked with an asterisk. BO = buffer overflow; UAF = use after free; ML = memory leak; AE = assertion error

<table>
<thead>
<tr>
<th>Prog</th>
<th>Type</th>
<th>AFL µTTE</th>
<th>AFLGo µTTE</th>
<th>FairFuzz µTTE</th>
<th>WindRanger µTTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>boringssl</td>
<td>UAF</td>
<td>1h13m</td>
<td>1h15m</td>
<td>1h35m</td>
<td>2h19m          +52m36s</td>
</tr>
<tr>
<td>guetzli</td>
<td>AE</td>
<td>6h31m</td>
<td>6h30m</td>
<td>6h30m</td>
<td>6h30m          +3h40m</td>
</tr>
<tr>
<td>libarchive</td>
<td>BO</td>
<td>5h33m</td>
<td>4h12m</td>
<td>5h21m</td>
<td>5h21m          +5h29m</td>
</tr>
<tr>
<td>libssh</td>
<td>ML</td>
<td>+4m41s</td>
<td>5m12s</td>
<td>5m46s</td>
<td>5m46s          +5m38s</td>
</tr>
<tr>
<td>libxml2</td>
<td>BO</td>
<td>4h41m</td>
<td>3h54m</td>
<td>4h13m</td>
<td>4h13m          +2h59m</td>
</tr>
<tr>
<td>pcre2</td>
<td>UAF</td>
<td>6h12m</td>
<td>5h52m</td>
<td>8h3m</td>
<td>8h3m           +5h23m</td>
</tr>
<tr>
<td>pcre2</td>
<td>BO</td>
<td>9h26m</td>
<td>8h35m</td>
<td>9h6m</td>
<td>9h6m           +6h39m</td>
</tr>
<tr>
<td>re2</td>
<td>BO</td>
<td>7h3m</td>
<td>8h28m</td>
<td>7h39m</td>
<td>7h39m          +5h49m</td>
</tr>
<tr>
<td>wolf2</td>
<td>BO</td>
<td>2h31m</td>
<td>1h35m</td>
<td>2h48m</td>
<td>2h48m          +1h13m</td>
</tr>
<tr>
<td>c-ares</td>
<td>BO</td>
<td>54s</td>
<td>+53s</td>
<td>1m6s</td>
<td>1m6s           +58s</td>
</tr>
<tr>
<td>openssl-1.1f</td>
<td>BO</td>
<td>5m38s</td>
<td>5m31s</td>
<td>6m1s</td>
<td>6m1s           +5m6s</td>
</tr>
</tbody>
</table>

| µTTE inc | +47%   | +30%   | +59%    |              |
| µTTE(+ 1h) inc | +66%   | +44%   | +77%    |              |

Table 4: TTT results of disabling each improved component in WindRanger. Short names for components: DC = distance calculation; SP = seed prioritization; M = data flow sensitive mutation; EES = explore-exploit stage switch

<table>
<thead>
<tr>
<th>Prog</th>
<th>Targets</th>
<th>Default Runs</th>
<th>µTTT</th>
<th>No DC Runs</th>
<th>µTTT</th>
<th>No SP Runs</th>
<th>µTTT</th>
<th>No M Runs</th>
<th>µTTT</th>
<th>No EES Runs</th>
<th>µTTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>imginfo</td>
<td>jpc_c_c.cpp:316</td>
<td>10</td>
<td>36m17s</td>
<td>50m18s</td>
<td>58m44s</td>
<td>41m20s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bmp_dec.c:474</td>
<td>10</td>
<td>1h2m</td>
<td>1h2m</td>
<td>1h19m</td>
<td>1h30m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jpeg_image.c:328</td>
<td>10</td>
<td>5h8m</td>
<td>5h48m</td>
<td>5h32m</td>
<td>5h29m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jas_stream.h:1026</td>
<td>9</td>
<td>1h37m</td>
<td>9</td>
<td>1h36m</td>
<td>1h17m</td>
<td>9h38m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lame</td>
<td>gain_analysis.c:224</td>
<td>10</td>
<td>22m40s</td>
<td>43m15s</td>
<td>53m14s</td>
<td>40m54s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bitstream.c:399</td>
<td>10</td>
<td>4h8m</td>
<td>4h27m</td>
<td>4h43m</td>
<td>3h54m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_audio.c:1452</td>
<td>10</td>
<td>8h51m</td>
<td>9h43m</td>
<td>10h18m</td>
<td>10h9m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mpglib_interface.c:412</td>
<td>8</td>
<td>1h57m</td>
<td>1h53m</td>
<td>1h44m</td>
<td>1h47m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mp42aac</td>
<td>Ap4ByteStream.cpp:199</td>
<td>10</td>
<td>3h11m</td>
<td>3h29m</td>
<td>3h41m</td>
<td>3h57m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ap4Link.h:172</td>
<td>10</td>
<td>6h1m</td>
<td>7h35m</td>
<td>7h15m</td>
<td>6h55m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ap4Channel.cpp:170</td>
<td>10</td>
<td>1h24m</td>
<td>1h24m</td>
<td>1h31m</td>
<td>1h97m</td>
<td>1h82m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mujs</td>
<td>jasun.c:1024</td>
<td>6</td>
<td>17m14m</td>
<td>17m36m</td>
<td>18h11m</td>
<td>18h28m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jsdumper.c:867</td>
<td>6</td>
<td>17m14m</td>
<td>17m36m</td>
<td>18h11m</td>
<td>18h28m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jsdumper.c:892</td>
<td>9</td>
<td>14h58m</td>
<td>15h45m</td>
<td>18h21m</td>
<td>18h57m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jsvalue.c:396</td>
<td>9</td>
<td>14h58m</td>
<td>15h45m</td>
<td>18h21m</td>
<td>18h57m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>objdump</td>
<td>objdump.c:10875</td>
<td>9</td>
<td>13h46m</td>
<td>13h45m</td>
<td>13h41m</td>
<td>13h97m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>section.c:943</td>
<td>10</td>
<td>1h15m</td>
<td>8h35m</td>
<td>8h49m</td>
<td>8h33m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>objdump.c:1514</td>
<td>6</td>
<td>17m49m</td>
<td>16h38m</td>
<td>18h42m</td>
<td>18h55m</td>
<td>18h11m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dwarflu.c:3176</td>
<td>6</td>
<td>17m49m</td>
<td>16h38m</td>
<td>18h42m</td>
<td>18h55m</td>
<td>18h11m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tcplib</td>
<td>print-ppp.c:729</td>
<td>10</td>
<td>3h20m</td>
<td>4h3m</td>
<td>4h5m</td>
<td>3h30m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>print-capture.h:591</td>
<td>10</td>
<td>5h84m</td>
<td>6h15m</td>
<td>6h31m</td>
<td>7h21m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>print-capture.h:840</td>
<td>10</td>
<td>12h33m</td>
<td>12h42m</td>
<td>12h40m</td>
<td>13h16m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>print-decode.c:832</td>
<td>4</td>
<td>21h30m</td>
<td>21h34m</td>
<td>21h14m</td>
<td>20h52m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

µTTT Inc | +12%   | +17%   | +22%   | +15%   |
based on AFL as baselines since WindRanger is also built on top of AFL. Furthermore, we also conduct fine-grained crash triage for the TTE experiments to acquire more accurate results.

6 RELATED WORK

Instead of discussing all the related works, we focus on the most related ones. Our work is mainly related to three topics: directed symbolic execution, directed grey-box fuzzing and coverage-guided grey-box fuzzing.

Directed Symbolic Execution. Directed Symbolic Execution (DSE) is a technique which relies on symbolic execution to reach the target sites. Several works have been proposed to deploy DSE in diverse tasks, such as patch testing [31, 35], to test critical system calls [19], to validate static analysis reports [11]. DSE leverages synergistic program information to gain directness, compared with general symbolic execution techniques like KLEE [6]. DSE can effectively test desired parts of PUT. However, DSE is not effective on real-world programs as symbolic execution techniques suffer from problems like path-explosion [37]. Compare to DSE, our solution WindRanger belongs to directed grey-box fuzzing (DGF) which is only rely on lightweight program analysis and is more effective on real-world programs as discussed in [3].

Directed Grey-box Fuzzing. Directed Grey-box Fuzzing (DGF) aims to bring directedness to grey-box fuzzing. AFLGo [3] is the first DGF. To gain the directness ability, the authors of AFLGo proposed methods to calculate the distance between a seed trace and target sites, this distance is used in the power schedule to decide the energy of the seed. Hawkeye [8] improves AFLGo with trace augmented power schedule, seed prioritization, and adaptive mutation. AFLGo and Hawkeye both consider all basic blocks in the seed distance calculation. Different from this, WindRanger focuses on important basic blocks which hinder inputs reaching target sites to improve the efficiency of DGF. Besides, FuzzGuard [49] leverages deep learning to filter out inputs which are unreachable to target sites before feeding them to DGF. However, WindRanger improves the efficiency of DGF and is orthogonal to FuzzGuard.

Some works deployed DGF on different kinds of target sites. Leopard [13] uses programs metrics to identify the vulnerable code in the program and set them as target sites. Parnesan [33] identifies the program locations instrumented by sanitizers (like ASAN [36]) and uses these locations as target sites. Besides, Parnesan calculates distance based on dynamically constructed CFG. CAFL [24] leverages information of ordered target sites and data conditions to boost up the DGF’s capability to expose targeted crashes. These works can gain better performance with more efficient DGF which WindRanger aims at.

Coverage-guided Grey-box Fuzzing. Coverage-guided Grey-box Fuzzing (CGF) aims to guide fuzzing to cover more code. AFL [23] is a classic CGF that employs coverage as feedback to guide seeds evolution. To increase the coverage in CGF, a research direction is to help CGF solve the condition constraints in the PUT. Hybrid fuzzing [37, 47] combines CGF and symbolic execution to enhance the constraint solving ability in CGF. Several other works adopt lightweight methods to help with constraint solving. Steelix [26] uses light-weight static analysis and binary instrumentation to gain the comparison progress information which helps CGF to explore paths protected by magic bytes comparisons. Angora [9] uses a novel technique called gradient descent based input search to solving constraints without symbolic execution. Greyone [16] adopts fuzzing-driven inference to conduct taint and uses data flow features to guide the evolution of fuzzing. FairFuzz [25] identifies and masks the bytes which influence producing inputs covering rare branches. Besides techniques for penetrating branches, there are also some other techniques focusing on improving the seed scheduling process [4, 28, 42, 46], detecting specific types of bugs [7, 41, 44] and automatic fuzz harness generation [1, 20, 48]. Although the aim of WindRanger is reaching target sites efficiently instead of increasing code coverage, WindRanger can benefit from some of the techniques used in CGF (e.g., the constraint solving techniques).

7 CONCLUSION

In this paper, we propose the concept of deviation basic block in the context of directed grey-box fuzzing. We propose an approach called WindRanger to effectively identify and use the deviation basic blocks to facilitate directed grey-box fuzzing. We conduct 38,400 CPU hours of experiments to evaluate WindRanger. The evaluation results show that WindRanger can outperform the state-of-the-art directed grey-box fuzzer by reaching the target sites 21% faster and exposing the vulnerabilities 44% faster. Last but not least, WindRanger shows practical usefulness by helping a security expert to find a 0-day vulnerability in an extensively fuzzed popular media library.

ACKNOWLEDGEMENTS

This research is partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR05-0001), NRF Investigatorship NRFI06-2020-0022-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001. This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. This work is also supported partially by the National Natural Science Foundation of China under Grants No. 61272078, 62032010, 62172201.

REFERENCES
