WindRanger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks

Zhengjie Du
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China

Yuekang Li[†]
Nanyang Technological University
Singapore

Yang Liu
Nanyang Technological University
Singapore

Bing Mao
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China

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 experiment 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.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; **Software and application security**;

KEYWORDS

Fuzz Testing, Directed Testing

ACM Reference Format:
Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *44th International Conference on Software Engineering (ICSE '22), May 21-29,2022,Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510197

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

[†]Corresponding author.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510197

[1] In this paper, we assume each target site corresponds to one basic block in the program

stressing unrelated program components. Thus, DGF can reach and test the target sites in the PUT faster than general purpose fuzzing techniques such as Coverage-guided Greybox Fuzzing (CGF), which focuses on maximizing the code coverage. This feature makes DGF excel in certain usage scenarios such as patch testing, crash reproduction and static analysis report verification [3]. For patch testing, the patched code sites can be set as the target sites and with the DGF technique, one can quickly test whether the patches are complete or not. For crash reproduction, the crashing site can be set as the target site and with the DGF technique, one can quickly reproduce and acquire the Proof-of-Concept crashing input. For static analysis report verification, the suspicious locations reported via static analyses can be set as the target sites and with the DGF technique, one can check whether the reported locations really contains vulnerabilities.

From these usage scenarios, we can observe that reaching the target sites is the primary goal for DGF. In practice, the time budgets for these usage scenarios are often limited and the number of target sites to reach can be huge. Therefore, speeding up the process of directing the fuzzer to reach the target sites can help to better fulfill these tasks. For example, for static analysis verification, if the DGF can reach more target sites with a fixed amount of time, then the chance of revealing vulnerabilities is increased. In recent years, researchers have proposed several techniques to allow DGF to reach and test more target sites with a limited time budget [8, 49].

In DGF, to gain directedness, the fuzzer needs to identify and prioritize the seeds [2] whose execution traces are *closer* to the target sites. Thus, correctly measuring the *distance* of a given seed towards the targets is an important factor for boosting the performance of DGF. To calculate the distance of a seed efficiently, the authors of AFLGo proposed a novel two-step approach: first, during static analysis, compute the distance between each basic block and the target sites on the call graph and control flow graph and then instrument the distance information into the program; during fuzzing, aggregate the distance values of each executed basic block as the distance of the seed. This approach enables AFLGo to calculate the seed distances with low runtime overhead.

Although AFLGo's approach of seed distance calculation is efficient and helps with the target orientation of the fuzzer, it suffers from two pitfalls: ❶ The seed distance calculated with *all* the basic blocks on the execution trace can be biased because not every block is the key of driving the execution towards the target sites. For example, suppose the fuzzer needs to evaluate two seeds, A and B. Assume seed A goes through a lot of basic blocks which are far away from the target site before it ends up somewhere very close to the target site, while seed B does not go through too many basic blocks and it ends up somewhere not as close. Rationally, the fuzzer should prioritize seed A rather than seed B since seed A can eventually get closer to the target site. However, if all the basic blocks are counted, the distance of seed A can be larger due to the noises introduced by some of the basic blocks. ❷ The seed distance calculation is *only* based on control flow information while the data flow information is ignored. Currently, the distance between a basic block and a target site is calculated based on the number of edges between them on the control flow graph and call graph. However, this distance cannot represent the difficulty of reaching the targets with the seed. For example, a basic block of a comparison for a checksum value can have two edges extending from it: one for the equal branch and one for the not equal branch. Fulfilling the edge of the equal branch is much harder than the not equal branch without a valid input. Thus, these two edges should be treated differently during distance calculation.

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. We name such basic blocks as *deviation basic blocks* (*DBB*s). If the execution trace at the *DBB*s can be altered, then the target site can be reached. Based on this observation, we propose WindRanger [3], a directed grey-box fuzzer augmented by *DBB* information. The workflow of WindRanger contains two major steps: static analysis and fuzzing. During static analysis, WindRanger first identifies all the potential *DBB*s which contain branches where the execution can deviate from the target sites. Then, WindRanger calculates the distance of the identified basic blocks to the target sites. During fuzzing, given a seed and its execution trace, WindRanger first selects the *DBB*s from the statically identified candidates for the seed based on its execution trace. (§ 3.1) Then, WindRanger calculates the distance value for the seed based on the *DBB*s and adjusts the distance value based on the condition matching difficulty of these *DBB*s (§ 3.3). With the calculated distances, WindRanger then prioritizes the seeds by adjusting the power scheduling strategy. In addition, bases on *DBB*s, WindRanger also improves the seed prioritization strategy (§ 3.5), mutation strategy (§ 3.4) as well as the explore-exploit switching decision process (§ 3.6) to further enhance the performance.

To evaluate the performance of WindRanger, we selected 29 programs from 3 datasets as benchmarks and used AFLGo as well as two coverage-guided grey-box fuzzers (AFL and FairFuzz) as baselines. We conducted 38,400 CPU hours of experiments to evaluate WindRanger's capability of reaching target sites in the programs and reproducing crashes. Our experiment results show that compare with AFLGo, AFL and FairFuzz, WindRanger can reach the target sites faster by 21%, 34%, and 37% and detect the target crashes faster by 44%, 66%, and 77% respectively. In practice, with manually selected target sites, WindRanger successfully discovered a 0-day vulnerability on one of the sites with a unique CVE ID assigned in ffmpeg [15].

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.

[2] 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

[3] WindRanger is a heroine in the game Dota2. Her ultimate skill is called Focus Fire, which allows her to quickly take down the target enemy.

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. [4] 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 $\frac{3+5+4+3+2+2+1}{7} = 2.86$, $\frac{3+2+2}{3} = 2.33$, and $\frac{3+2+1}{3} = 2$ 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 *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 *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 because 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 *deviation basic blocks* (*DBB*s).

Our Approach. If we calculate the distances by averaging only the *DBB*s, 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 *DBB*s, the problem now becomes how to identify them.

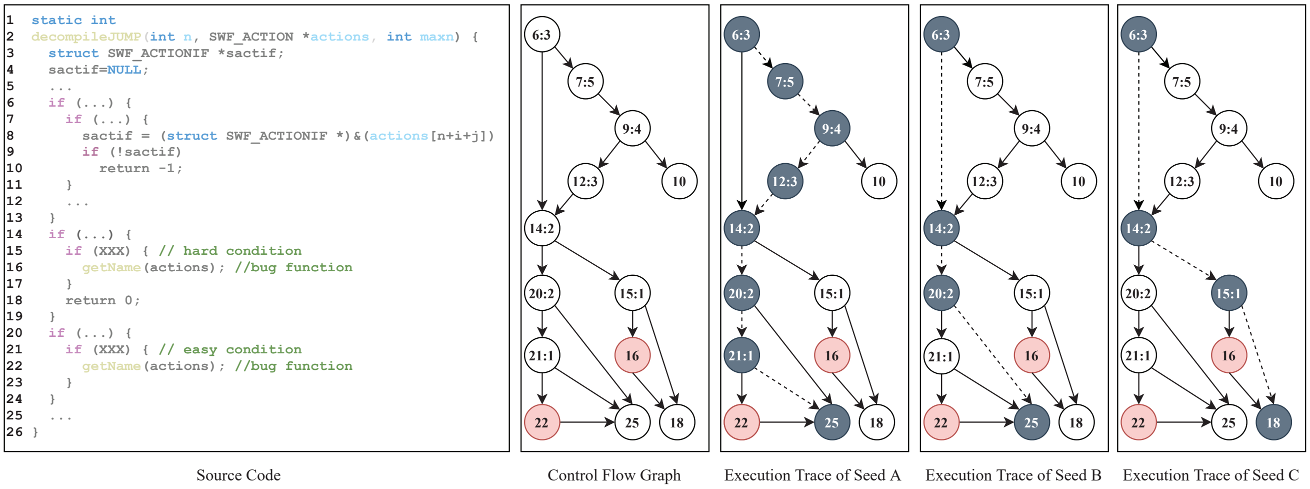
Although in this example, all the *DBB*s happen to be the basic blocks with the shortest distance, the identification of them is not that straightforward. Suppose we have another seed D, whose execution trace is <6:3 → 7:5 → 9:4 → 10>. For this seed, even though node 6:3 has a shorter distance than node 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 *DBB*s in the context of DGF and propose an approach to identify them on execution traces. Furthermore, we propose an approach to leverage the *DBB*s 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 (*DBB*s), 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

[4] In [3] [8], harmonic means are used to balance the distance between multiple target sites. In WindRanger, we focus on improving the seed distance calculation, so we use a naive approach to calculate the basic block distance in this example for simplicity.



```
1  static int
2  decompileJUMP(int n, SWF_ACTION *actions, int maxn) {
3    struct SWF_ACTIONIF *sactif;
4    sactif=NULL;
5    ...
6    if (...) {
7      if (...) {
8        sactif = (struct SWF_ACTIONIF *)&(actions[n+i+j])
9        if (!sactif)
10         return -1;
11     }
12     ...
13   }
14   if (...) {
15     if (XXX) { // hard condition
16       getName(actions); //bug function
17     }
18     return 0;
19   }
20   if (...) {
21     if (XXX) { // easy condition
22       getName(actions); //bug function
23     }
24   }
25   ...
26 }
```

Source Code Control Flow Graph Execution Trace of Seed A Execution Trace of Seed B Execution Trace of Seed C

Figure 1: Code snippet crafted from CVE-2018-8962, its corresponding CFG and execution traces of 3 different seeds

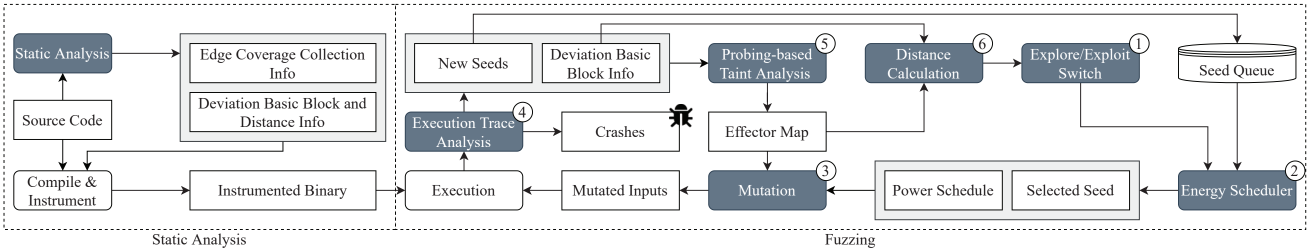


Figure 2: The overall workflow of WindRanger

(⑤). The mutation helps to steer the fuzzer to pass the conditionals of the *DBB*s. 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 *DBB*s (④). Before adding the seed to the seed queue, WindRanger will calculate its distance towards the target sites according to the *DBB*s (⑥) 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: ❶ how does WindRanger identify the *DBB*s for a seed using static and dynamic analyses (§ 3.1); ❷ how does WindRanger perform the probing based taint analysis (§ 3.2); ❸ how does WindRanger calculate the seed distance and fine-tune it with the taint analysis result (§ 3.3); ❹ how does WindRanger mutate the input with the taint analysis result (§ 3.4); ❺ how does WindRanger use the *DBB*s for seed prioritization (§ 3.5); ❻ 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 *DBB*s in the execution trace of a seed, WindRanger first finds the potential *DBB*s in the program with static analysis. Then, during fuzzing, WindRanger pinpoints the *DBB*s in the execution trace with analyses on executed basic blocks and their relations with the potential *DBB*s.

Static *DBB* Collection. Here we provide the definition of the potential *DBB*s for a program and then discuss how WindRanger finds the potential *DBB*s with static analysis.

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

$$\Phi(P) = \{b \in AllBB(P) \mid isReachable(b, T) \wedge \\ (\exists c \in Successors(b))[\neg isReachable(c, T)]\} \tag{1}$$

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 *DBB*s 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 *DBB*s of an execution trace and then discuss how WindRanger identifies the *DBB*s dynamically.

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

$$\Phi(s) = \{b \in \xi(s) \mid b \in \Phi(P) \wedge (\forall c \in ReachableSucc(b))[c \notin \xi(s)]\} \tag{2}$$

where $\xi(s)$ denotes all basic blocks in the execution trace of the seed s; $ReachableSucc(b)$ 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 *DBB*s of a seed are the basic blocks in its execution traces which are the potential *DBB*s 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 *DBB*s 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 *DBB*s for a program are used as intermediate results to acquire the *DBB*s 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 $ValueOf(var, s)$ returns the value of variable 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 ($\mathbb{VAR}$). Then, WindRanger mutates the seed byte by byte with a set of predefined mutation operators ($\mathbb{OP}$). 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 predefine three mutation operations for $\mathbb{OP}$: 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.

Algorithm 1 Effector Map Inference

```
Input: $s$, the seed to be probed
Input: $\mathbb{OP}$, the predefined mutation operators
Input: $P'$, the instrumented program
Output: $M$, the effector map of $s$
$InitWithEmptyMap(M)$
$\xi(s) \leftarrow Execute(s, P')$
$\mathbb{VAR} \leftarrow ExtractConstraintVariables(\xi(s))$
for $0 \leq pos < |s|$ do
    for $op \in \mathbb{OP}$ do
        $s' \leftarrow Mutate(s, pos, op)$
        for $var \in \mathbb{VAR}$ do
            if $ValueOf(var, s) \neq ValueOf(var, s')$ then
                $M[var] = M[var] \cup \{pos\}$
```

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:

$$d_s(s, T_b) = \frac{\sum_{m \in \Phi(s)} d_b(m, T_b)}{|\Phi(s)|} \tag{3}$$

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

Note that structure of Equation 3 and the definition of $d_b(m, T_b)$ are both the same as in AFLGo [3] and HawkEye [8] except that we use the *DBB*s 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:

$$d_s(s, T_b) = \frac{\sum_{m \in \Phi(s)} d_b(m, T_b) \cdot \Psi(s, m)}{|\Phi(s)|} \tag{4}$$

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:

$$\Psi(s, m) = min(Max_{\Psi}, \lceil \frac{NumOfEffectiveBytes(s, m)}{\gamma} \rceil) \tag{5}$$

where $NumOfEffectiveBytes(s, m)$ indicates the number of input bytes that can affect the constraint variable(s) of m according to the effector map of s; γ is a constant ratio which controls the granularity; Max_Ψ 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 `DEADBEEF` and the md5sum of the input is `HEADBEEF`. 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 *DBB*s 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 *DBB*s.

Specifically, during fuzzing, WindRanger maintains all existing *DBB*s as a global set (denoted as $\Phi(F)$). In the exploit stage, when all *DBB*s 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 $hit(x, b)$) or not can be defined as

$$hit(x,b) = \begin{cases} 1, & b \in \xi(x) \\ 0, & b \notin \xi(x) \end{cases} \tag{6}$$

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

$$numHit(b) = \sum_{x \in I} hit(x, b) \tag{7}$$

where I 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

$$numHit(b) > \upsilon \cdot \min_{b' \in T} numHit(b') \tag{8}$$

where T is the set of all basic blocks WindRanger has encountered during fuzzing; υ 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 ($\hat{A}_{12}$) [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 [12] 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: ❶ 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; ❷ Then, we calculate the average first-reach time across the 10 runs for each basic block; ❸ 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, there exists no basic block that happens to be reached exactly at the scale time values. Under this circumstance, we choose the target sites whose first-reach time is close to the scale time value.

With the benchmark built on top of UniFuzz, we compare WindRanger with the baselines to evaluate how much time they cost to reach the selected target sites. We conducted 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* (AFL,FairFuzz) 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 $\hat{A}_{12}$ 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

***The detailed experiment results for all the programs are on our website [14]**

Prog	Targets	AFL				AFLGo				FairFuzz				WindRanger	
		Runs	μTTT	Factor	$\hat{A}_{12}$	Runs	μTTT	Factor	$\hat{A}_{12}$	Runs	μTTT	Factor	$\hat{A}_{12}$	Runs	μTTT
imginfo	jpc_cs.c:316	10	1h28m	2.43	**0.87**	10	1h1m	1.68	**0.82**	10	1h17m	2.14	**0.85**	10	*36m17s
	bmp_dec.c:474	10	2h20m	2.23	**0.86**	10	1h40m	1.61	**0.75**	10	2h12m	2.11	**0.79**	10	*1h2m
	jas_image.c:378	10	6h30m	1.26	0.70	10	7h42m	1.50	**0.80**	10	5h51m	1.14	0.61	10	*5h8m
	jas_stream.h:1026	7	16h33m	1.42	**0.82**	8	15h24m	1.35	**0.73**	7	18h0m	1.54	**0.86**	9	*11h37m
lame	gain_analysis.c:224	10	1h10m	3.11	**0.90**	10	51m35s	2.30	**0.88**	10	1h17m	3.40	**0.94**	10	*22m40s
	bitstream.c:399	10	5h16m	1.27	0.69	10	*3h59m	0.96	0.47	10	5h39m	1.40	**0.75**	10	4h8m
	get_audio.c:1452	10	14h40m	1.70	**0.88**	10	11h46m	1.33	**0.79**	10	13h1m	1.47	**0.84**	10	*8h51m
	mpglib_interface.c:142	7	19h37m	1.64	**0.77**	8	15h53m	1.43	0.64	8	18h13m	1.52	**0.71**	9	*11h57m
mp42aac	Ap4ByteStream.cpp:199	10	4h43m	1.47	**0.76**	10	4h12m	1.31	0.66	10	5h25m	1.69	**0.77**	10	*3h11m
	Ap4Lish.h:172	10	9h38m	1.34	**0.74**	10	*7h1m	0.97	0.50	10	9h14m	1.28	**0.75**	10	7h11m
	Ap4CttsAtom.cpp:170	8	15h48m	1.46	**0.81**	8	14h31m	1.34	**0.75**	10	15h22m	1.42	**0.80**	**9**	**10h47m**
	Ap4ElstAtom.cpp:100	5	20h10m	1.17	0.66	6	18h38m	1.08	0.59	3	21h43m	1.26	**0.73**	7	*17h14m
mujs	jsrun.c:1024	10	5h39m	1.47	0.72	10	5h11m	1.35	0.68	10	7h13m	1.66	0.81	10	*3h50m
	jsdump.c:867	9	10h12m	2.18	**0.85**	10	4h59m	1.15	**0.68**	10	7h13m	1.66	**0.81**	10	*4h21m
	jsdump.c:892	9	14h1m	1.85	**0.86**	10	9h23m	1.24	0.64	8	13h11m	1.74	**0.82**	10	*7h33m
	jsvalue.c:396	6	19h10m	1.28	**0.75**	8	17h44m	1.18	**0.75**	7	18h55m	1.26	**0.76**	9	*14h58m
objdump	objdump.c:10875	10	5h6m	1.30	0.69	10	4h38m	1.18	0.61	10	6h13m	1.58	**0.81**	10	*3h55m
	section.c:943	10	10h8m	1.20	**0.74**	10	11h29m	1.36	**0.84**	10	11h50m	1.40	**0.85**	10	*8h24m
	objdump.c:1514	8	15h31m	1.12	0.61	10	*12h54m	0.93	0.47	8	16h31m	1.20	0.65	9	13h46m
	dwarf2.c:3176	3	20h58m	1.17	0.65	4	19h59m	1.12	0.61	5	19h19m	1.08	0.56	6	*17h49m
tcpdump	print-ppp.c:729	10	5h2m	1.51	0.69	10	4h35m	1.37	0.63	10	5h29m	1.64	**0.77**	10	*3h20m
	extract.h:591	9	9h53m	1.70	**0.76**	10	7h34m	1.40	**0.69**	10	8h24m	1.44	**0.74**	10	*5h48m
	print-l2tp.c:840	10	15h4m	1.20	**0.70**	10	14h27m	1.15	0.63	10	15h21m	1.22	**0.70**	10	*12h33m
	print-decnet.c:832	5	20h23m	0.95	0.44	3	22h37m	1.05	0.58	6	*19h53m	0.92	0.40	4	21h39m
μTTT inc		**+56%**				**+31%**				**+55%**					
mean $\hat{A}_{12}$		**0.75**				**0.67**				**0.75**					
μTTT inc (all programs)		**+34%**				**+21%**				**+37%**					

confidence to conclude that WindRanger has better capability to reach the given target sites than other tools.

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 (μTTE > 1 hour), WindRanger significantly outperforms other tools by 1.22× to 2.2× faster to expose them. For CVE-2016-4491 and CVE-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 CVE-2018-8807 and CVE-2018-8962, of which $\hat{A}_{12}$ is equal or greater than 0.75 against other tools. 3) For CVE-2016-4489 and CVE-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 μTTE.

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 each program only contains one bug. The experiment results are shown in Table 3. The results in Table 3 demonstrate that by setting vulnerabilities locations as target sites, *DGF*s (WindRanger and AFLGo) can achieve better performance on exposing relative crashes than *CGF*s (AFL and FairFuzz). Among the 11 vulnerabilities, WindRanger gains better μTTE 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,

Table 2: Crash exposure results on aflgo test suite. For each CVE, the statistically significant (*p-value* < 0.05) values of $\hat{A}_{12}$ is highlighted in bold; the shortest mean TTE is marked with an asterisk

	CVE-ID	Tool	Runs	μTTE	Factor	$\hat{A}_{12}$
binutils	2016-4487	WindRanger	20	∗1m58s	-	-
		AFL	20	3m41s	1.92	**0.82**
		AFLGo	20	3m18s	1.37	0.65
		FairFuzz	20	4m15s	2.22	**0.87**
	2016-4488	WindRanger	20	∗10m25s	-	-
		AFL	20	16m39s	1.60	**0.86**
		AFLGo	20	13m15s	1.27	**0.70**
		FairFuzz	20	15m57s	1.50	**0.86**
	2016-4489	WindRanger	20	5m7s	-	-
		AFL	20	6m38s	1.30	0.66
		AFLGo	20	∗4m22s	0.85	0.39
		FairFuzz	20	5m19s	1.04	0.49
	2016-4490	WindRanger	20	2m5s	-	-
		AFL	20	∗1m21s	0.65	0.25
		AFLGo	20	1m29s	0.71	0.30
		FairFuzz	20	2m48s	1.34	0.67
	2016-4491	WindRanger	6	∗5h26m	-	-
		AFL	5	6h37m	1.22	0.63
		AFLGo	4	7h4m	1.30	**0.70**
		FairFuzz	3	7h28m	1.37	**0.72**
	2016-4492	WindRanger	20	∗8m32s	-	-
		AFL	20	14m34s	1.71	**0.84**
		AFLGo	20	9m39s	1.13	0.63
		FairFuzz	20	15m4s	1.77	**0.89**
	2016-6131	WindRanger	8	∗4h58m	-	-
		AFL	3	7h17m	1.46	**0.82**
		AFLGo	6	6h5m	1.22	0.66
		FairFuzz	5	6h57m	1.40	**0.76**
libming	2018-8807	WindRanger	20	∗1h32m	-	-
		AFL	20	2h16m	1.47	**0.78**
		AFLGo	20	2h41m	1.73	**0.85**
		FairFuzz	20	2h49m	1.82	**0.87**
	2018-8962	WindRanger	20	∗1h38m	-	-
		AFL	20	2h22m	1.46	**0.75**
		AFLGo	20	3h36m	2.20	**0.93**
		FairFuzz	20	2h50m	1.74	**0.80**

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 μTTE exceed 1 hour, WindRanger can achieve larger performance gain with speed up of 66% against AFL, 44% against AFLGo and 77% against FairFuzz.

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.

4.5 New Vulnerability (RQ4)

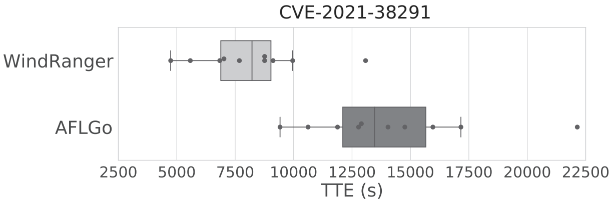


Figure 3: The time cost to expose the new vulnerability in 10 runs

To check the if WindRanger can be used for hunting previously unknown bugs, we gave the prototype of WindRanger to a security expert [5]. 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 (TTE) 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× smaller than the meadian 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 *DBB*s. 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

[5] The security expert works in a security company and is not involved in the development of our tool.

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 $\hat{A}_{12}$ 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

Prog	Type	AFL			AFLGo			FairFuzz			WindRanger
		μTTE	Factor	$\hat{A}_{12}$	μTTE	Factor	$\hat{A}_{12}$	μTTE	Factor	$\hat{A}_{12}$	μTTE
boringssl	UAF	1h47m	2.04	**0.86**	1h21m	1.56	**0.82**	1h55m	2.19	**0.85**	*52m36s
guetzli	AE	6h13m	1.29	0.65	5h58m	1.24	0.59	6h30m	1.34	0.67	*3h40m
libarchive	BO	5h33m	1.59	**0.79**	4h12m	1.35	0.67	5h21m	1.53	**0.75**	*3h29m
libssh	ML	*4m41s	0.83	0.37	5m12s	0.92	0.47	5m46s	1.02	0.56	5m38s
libxml2	BO	4h41m	1.57	**0.78**	3h54m	1.30	**0.67**	5h16m	1.76	**0.77**	*2h59m
pcre2	UAF	6h12m	1.82	**0.81**	5h52m	1.72	**0.74**	8h3m	2.37	**0.93**	*3h23m
pcre2	BO	9h26m	1.41	**0.71**	8h38m	1.29	0.67	9h6m	1.36	**0.70**	*6h39m
re2	BO	7h2m	1.20	**0.67**	8h29m	1.45	**0.76**	7h39m	1.31	**0.59**	*5h49m
woff2	BO	2h51m	2.34	**0.87**	1h55m	1.57	0.65	2h48m	2.29	**0.74**	*1h13m
c-ares	BO	54s	0.93	0.45	*53s	0.91	0.42	1m6s	1.13	0.59	58s
openssl-1.0.1f	BO	5m38s	1.10	0.57	5m31s	1.08	0.57	6m1s	1.17	0.65	*5m6s
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

Prog	Targets	Default		No DC		No SP		No M		No EES	
		Runs	μTTT	Runs	μTTT	Runs	μTTT	Runs	μTTT	Runs	μTTT
imginfo	jpc_cs.c:316	10	36m17s	10	43m57s	10	50m18s	10	58m44s	10	41m20s
	bmp_dec.c:474	10	1h2m	10	1h21m	10	1h36m	10	1h19m	10	1h30m
	jas_image.c:378	10	5h8m	10	5h48m	10	5h22m	10	5h32m	10	5h29m
	jas_stream.h:1026	9	11h37m	9	12h48m	9	13h6m	9	14h17m	9	13h8m
lame	gain_analysis.c:224	10	22m40s	10	33m18s	10	43m15s	10	53m14s	10	40m54s
	bitstream.c:399	10	4h8m	10	4h27m	10	4h43m	10	4h19m	10	3h54m
	get_audio.c:1452	10	8h51m	10	9h43m	10	10h18m	10	10h36m	10	10h9m
	mpglib_interface.c:142	8	11h57m	8	13h5m	8	12h44m	8	14h39m	8	14h7m
mp42aac	Ap4ByteStream.cpp:199	10	3h11m	10	3h29m	10	3h41m	10	4h3m	10	3h57m
	Ap4Lish.h:172	10	7h11m	10	7h33m	10	7h5m	10	7h15m	10	6h55m
	Ap4CttsAtom.cpp:170	9	10h47m	9	12h7m	8	13h10m	8	13h43m	8	13h19m
	Ap4ElstAtom.cpp:100	6	17h14m	6	17h51m	6	17h36m	6	18h11m	6	18h28m
mujs	jsrun.c:1024	10	3h50m	10	4h46m	10	4h24m	10	4h53m	10	4h36m
	jsdump.c:867	10	4h21m	10	4h52m	10	5h18m	10	5h33m	10	5h35m
	jsdump.c:892	10	7h33m	10	9h16m	10	8h49m	10	9h17m	10	8h20m
	jsvalue.c:396	9	14h58m	9	16h47m	9	15h45m	8	17h21m	8	16h57m
objdump	objdump.c:10875	10	3h55m	10	4h29m	10	4h11m	10	4h53m	10	4h29m
	section.c:943	10	8h24m	10	9h10m	10	8h54m	10	8h49m	10	8h33m
	objdump.c:1514	9	13h46m	9	13h45m	9	12h47m	9	13h21m	9	13h6m
	dwarf2.c:3176	6	17h49m	6	18h28m	6	18h42m	6	18h55m	6	18h11m
tcpdump	print-ppp.c:729	10	3h20m	10	3h38m	10	4h13m	10	4h5m	10	3h30m
	extract.h:591	10	5h48m	10	6h15m	10	7h13m	10	6h31m	10	7h21m
	print-l2tp.c:840	10	12h33m	10	12h42m	10	13h40m	10	12h54m	10	13h16m
	print-decnet.c:832	4	21h30m	4	19h44m	4	21h34m	4	21h14m	4	20h52m
	μTTT Inc				**+12%**		**+17%**		**+22%**		**+15%**

pointer analysis. After analyzing the experiment results, we empirically find that the quality of the current reachability analysis is enough to yield good performance. Thus, we leave the improvement of the reachability analysis as future work.

The second internal threat comes from the configurable options in WindRanger such as the variable for tuning the seed distances and the threshold for the exploration-exploitation switching. We empirically choose the variable values for experiments and the current experiment results are promising. We believe fine-tuning the configurable options can further improve the experiment results but it is not the key technique to discuss. Thus, we leave fine-tuning the configurable options as future work.

The external threats mainly come from the experiment setup. Tool evaluation results should always be taken with a grain of salt. To mitigate the randomness in the experiments, we follow the suggestions in [22] to repeat the experiments for 10 or 20 runs and apply statistical tests on the results. To mitigate the influences of different implementations, we choose the techniques implemented 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. Parmesan [33] identifies the program locations instrumented by sanitizers (like ASAN [36]) and uses these locations as target sites. Besides, Parmesan 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 Greybox 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-NCR005-0001), 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

[1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. [n.d.]. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[2] GUN Binutils. 1990. A collection of binary tools. https://www.gnu.org/software/binutils/

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

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

[5] Marcel Böhme. 2021. Directed Greybox Fuzzing with AFL. https://github.com/aflgo/aflgo

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[7] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

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

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

[10] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1967–1983.

[11] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*. 144–155.

[12] Docker. 2021. Use containers to Build, Share and Run your applications. https://www.docker.com/resources/what-container

[13] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 60–71.

[14] Zhengjie Du and Yuekang Li. 2021. Windranger: Advanced Directed Greybox Fuzzer. https://sites.google.com/view/windranger-directed-fuzzing/

[15] FFmpeg. 2021. A complete, cross-platform solution to record,convert and stream audio and video. https://www.ffmpeg.org/

[16] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[17] Google. 2014. FFmpeg and a thousand fixes. https://security.googleblog.com/2014/01/ffmpeg-and-thousand-fixes.html

[18] Google. 2021. Set of tests for fuzzing engines. https://github.com/google/fuzzer-test-suite

[19] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium (USENIX Security 13)*. 49–64.

[20] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. [n.d.]. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[21] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 249–263.

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[23] lcamtuf. 2021. American Fuzzy Lop (AFL) Fuzzer. https://lcamtuf.coredump.cx/afl/

[24] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*.

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

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.

[27] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*.

[28] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/3338906.3338975

[29] Libming. 2008. C library for generating Macromedia Flash files (.swf). http://www.libming.org/

[30] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140

[31] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 235–245.

[32] NVD. 2018. CVE-2018-8962. https://nvd.nist.gov/vuln/detail/CVE-2018-8962

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[34] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[35] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 218–227.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIXATC 12)*. 309–318.

[37] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[38] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[39] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[40] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010.

[41] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

[42] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *NDSS*.

[43] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[44] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 765–777. https://doi.org/10.1145/3377811.3380396

[45] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 712–723.

[46] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[47] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[48] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2811–2828. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen

[49] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*. 2255–2269.

[50] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong