SELECTFUZZ: Efficient Directed Fuzzing with Selective Path Exploration

Changhua Luo
Chinese University of Hong Kong
Hong Kong SAR, China
chluo@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Penghui Li
Chinese University of Hong Kong
Hong Kong SAR, China
phli@cse.cuhk.edu.hk

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179296

***Abstract*—Directed grey-box fuzzers specialize in testing specific target code. They have been applied to many security applications such as reproducing known crashes and detecting vulnerabilities caused by incomplete patches. However, existing directed fuzzers favor the inputs discovering new code regardless whether the newly uncovered code is relevant to the target code or not. As a result, the fuzzers would extensively explore irrelevant code and suffer from low efficiency.**

In this paper, we distinguish relevant code in the target program from the irrelevant one that does not help trigger the vulnerabilities in target code. We present SELECTFUZZ, a new directed fuzzer that selectively explores relevant program paths for efficient crash reproduction and vulnerability detection. It identifies two types of relevant code—path-divergent code and data-dependent code, that respectively captures the control- and data- dependency with the target code. It then selectively instruments and explores only the relevant code blocks. We also propose a new distance metric that accurately measures the reaching probability of different program paths and inputs.

We evaluated SELECTFUZZ with real-world vulnerabilities in sets of diverse programs. SELECTFUZZ significantly outperformed a baseline directed fuzzer by up to 46.31×, and performed the best in the Google Fuzzer Test Suite. Our experiments also demonstrated that SELECTFUZZ and the existing techniques such as path pruning are complementary. Finally, with SELECTFUZZ, we detected 14 previously unknown vulnerabilities—including 6 new CVE IDs—in well tested real-world software. Our report has led to the fix of 11 vulnerabilities.

I. INTRODUCTION

Directed grey-box fuzzing has gained much traction for its high effectiveness in patch testing [1–3], crash reproduction [4–6], vulnerability verification [7–9], *etc.* Unlike general coverage-guided fuzzers that explore all the code in a target program, directed fuzzers test the program in direction(s) towards some specific code location(s). Depending on the application scenarios, the fuzzing targets can be derived from bug reports [5, 10, 11], patch change logs [2, 12], and results of static analysis tools [13–15], *etc.* Past practices [2, 16, 17] have shown that directed fuzzers can efficiently validate vulnerabilities and reproduce crashes.

The existing directed fuzzers (*e.g.*, AFLGo [16]) are usually built upon AFL [18] and their path exploration strategy follows that of AFL. Specifically, directed fuzzers generate inputs using the code coverage feedback. They then favor the inputs that trigger new coverage. In this work, we argue that this path exploration strategy could severely limit the performance of directed fuzzers. This is because it would guide directed fuzzers to explore mostly *irrelevant* code, by visiting which the directed fuzzers make almost no progress in reaching the targets or triggering the designated vulnerabilities. Our experiments on real-world programs showed that many fuzzing inputs indeed triggered only the irrelevant code; we name such inputs as *irrelevant inputs*. This observation is consistent with recent results that the executions of 65.1% of fuzzing inputs do not help reach the targets [17]. Even if the executions reach the targets, it has been shown that not all triggered code is critical in exploitation [19]. Moreover, the exploration of irrelevant code is likely to lead the directed fuzzers to continue generating inputs uncovering mostly irrelevant code, which further limits their efficiency.

The existing directed fuzzers could not effectively exclude the irrelevant inputs and thus waste much energy on exploring irrelevant code. Specifically, most directed fuzzers do not attempt to distinguish irrelevant inputs (resp. code) from other inputs (resp. code). They merely consider the distance to the target code when allocating energy to inputs [1, 6, 12, 16, 20, 21]. Because of the large number of irrelevant inputs in the input queue, they would unavoidably allocate excessive energy to the irrelevant inputs and explore irrelevant code, resulting in low efficiency. To improve the fuzzing efficiency, Beacon—a state-of-the-art directed fuzzer—terminates the execution of paths unreachable to the targets [2]. This allows it to exclude the inputs that discover only unreachable code and thus decrease the number of irrelevant inputs in the input queue. However, as we will show in §III-A, reachable code might also be irrelevant to the target code. Our evaluation results further reveal that a large proportion (*i.e.*, 87.67%) of reachable code in real-world programs is actually irrelevant. Consequently, even Beacon narrows the fuzzing scope to reachable code, it still wastes much energy on exploring *reachable irrelevant code*.

This work aims to improve the state-of-the-art directed fuzzing techniques by avoiding exploring irrelevant code. A major challenge we face is to precisely identify the relevant code a directed fuzzer should explore. On the one hand, without identifying the relevant code, the directed fuzzer would not distinguish important feedback provided by the relevant code from that provided by other code, then is unable to prioritize the inputs. On the other hand, if we mistakenly identify the relevant code, the directed fuzzer would be misled to explore

DOI 10.1109/SP46215.2023.00060

irrelevant code and suffer from low efficiency.

The second challenge we encounter is how to exclude the irrelevant code from directed fuzzing. An intuitive approach is removing the irrelevant code from the target program using techniques like software debloating [22, 23]. However, it would usually cause runtime errors before the executions reach the target code. Even the fuzzer could generate inputs that reach the target code and trigger the vulnerabilities, the inputs might not apply to the original program. Beacon [2] avoids some unnecessary path exploration by inserting assertions to early terminate the executions. However, this approach does not exclude the reachable irrelevant code.

To address the first challenge, we identify code relevant to a target code location by analyzing its control and data dependencies with the target code. Our key insight is that triggering a vulnerability requires satisfying certain control-flow and data-flow conditions related to the target code. Specifically, we identify the *path-divergent code* and the *data-dependent code* as relevant code, on which the target code is control- and data- dependent, respectively. Path-divergent code provides important feedback for efficiently *reaching* the target code (*i.e.*, control-flow conditions). Data-dependent code guides the fuzzer to explore the target code with different data values (*i.e.*, data-flow conditions); it thus assists the *exploitation* stage.

We selectively instrument only relevant code to address the second challenge. By instrumenting only relevant code the fuzzer receives feedback from only the relevant code, though it might also execute numerous irrelevant code blocks from which it receives no feedback. It thus would not favor the inputs uncovering only the irrelevant code as such code is not visible to the fuzzer. Consequently, the fuzzer has no incentive in exploring the irrelevant code, which in turn allows it to allocate its most energy to test the relevant code. This also avoids the potential runtime errors that might result from software debloating-based techniques. Besides, not instrumenting irrelevant code would reduce the runtime instrumentation overhead, which also affects the fuzzing efficiency. The AFL-based fuzzers incur over 30% overhead [24, 25]; the directed fuzzers introduce even higher runtime overheads because of the additional distance measurement. Since we instrument only a small fraction of basic blocks, the instrumentation overhead could be greatly reduced.

In this work, we present SELECTFUZZ, an efficient directed fuzzer that employs selective path exploration of only the code relevant to the fuzzing targets. In the first step, SELECTFUZZ statically identifies the relevant code with respect to the targets. To identify the path-divergent code, it needs to determine if one path can reach the targets. To this end, we propose a new distance metric to infer if a code block is reachable to the targets. We also improve the existing input prioritization mechanism with the new distance metric. Prior works prefer the shortest paths and might not favor some inputs whose executions are (very) likely to reach the target code. SELECTFUZZ instead favors the inputs that are more likely to reach the targets even the code paths are long. In the second step, SELECTFUZZ applies coverage and distance instrumentations to only the relevant code. This allows it to keep progressing towards the targets instead of being distracted to explore the irrelevant code paths.

We evaluated SELECTFUZZ on two datasets: 1) a set of known vulnerabilities that have been evaluated in prior works [1, 2, 16], and 2) the Google Fuzzer Test Suite (GFTS) [26]. Our evaluation results demonstrate SELECTFUZZ's high efficiency in reproducing known crashes. In the first experiment, SELECTFUZZ identified and instrumented 1.96% of total basic blocks (*i.e.*, 12.33% of the reachable basic blocks) as relevant. It outperformed AFLGo by triggering 7 more vulnerabilities and achieving a speedup of up to 46.31×. We further compared SELECTFUZZ with a closely related work Beacon [2] and found that they exhibited different improvements and were complementary. In the experiment on comparing with other complementary fuzzers using the GFTS, SELECTFUZZ performed the best in 8 out of 17 cases. Finally, we applied SELECTFUZZ to test the completeness of vulnerability patches. Interestingly, it discovered 14 new vulnerabilities—including 6 new CVE IDs—in the software that has been heavily tested by prior works. At the time of writing, 13 vulnerabilities have been confirmed and 11 have been promptly fixed.

In summary, this paper makes the following contributions:

- We propose a new concept of (ir)relevant code and demonstrate its importance in directed fuzzing.
- We present and implement a novel directed fuzzer, SELECTFUZZ, that enhances the efficiency of directed fuzzing with selective path exploration.
- SELECTFUZZ achieved high performance and can well complement existing approaches such as Beacon.
- With SELECTFUZZ, we detected 14 new vulnerabilities.

II. BACKGROUND

In this section, we introduce directed greybox fuzzing in §II-A and summarize prior works on improving directed fuzzing efficiency in §II-B.

A. Directed Greybox Fuzzing

Different from conventional fuzzers that explore the whole program state space, directed fuzzers are designed to thoroughly test only a target part of the program [2, 16]. Directed fuzzing has many applications. First, it can be used to reproduce known crashes and generate one-day exploits [5, 13]. Besides, it could work with various heuristics (*e.g.*, sanitizer guided [27], memory-function guided [28, 29], function-complexity guided [30], and commits guided [2, 16, 31] approaches *etc.*) or other security analysis techniques (*e.g.*, static program analysis [8, 9, 14, 15], concolic execution [13, 32, 33], *etc.*) to efficiently find new vulnerabilities. Since real-world programs are usually very large and complex, prioritizing testing on the potential buggy code could improve the vulnerability detection efficiency in fuzzing. Indeed, quite a few recent works adopted directed fuzzing and found many new vulnerabilities in real-world programs [2, 15, 27, 30, 31].

B. Improving Directed Fuzzing Efficiency

There are two directions towards improving directed fuzzing efficiency. First, researchers have leveraged various techniques (*e.g.*, symbolic executions [33] or taint tracking [27]) to efficiently *generate high-quality inputs*. Second, a lot of works have attempted to *identify* the inputs interesting to directed fuzzers [1, 2, 6, 12, 16, 17, 21]. Our work focuses on the second aspect, similar to AFLGo and Beacon,

Identifying and favoring interesting inputs are important since overly testing uninteresting inputs would significantly waste the fuzzing efforts. In the following, we investigate how prior directed fuzzers identify interesting inputs and categorize them into two types—distance-based input prioritization and input reachability analysis.

1) Distance-based Input Prioritization: A few works defined distance metrics to identify interesting inputs in directed fuzzing. AFLGo is the first work that systematically studied directed fuzzing and it defined a distance metric to assess the goodness of an input [16]. The distance between two basic blocks is calculated on the control flow graph using the Dijkstra's shortest path algorithm. If the two blocks are in different functions, AFLGo adds a pre-defined constant C to estimate the cross-function distance. For instance, if $b1$ is a call site of a function $f()$, then $d_{bb}(b1, b) = C, \forall b \in f()$. The input distance is defined as the average distance of all basic blocks visited by the input to the target blocks. A series of follow-up works improved AFLGo using either precise program analysis techniques or new distance metrics [1, 6, 12, 21, 34]. However, their key ideas are similar—they aim to identify interesting seed inputs based on their distance metrics and assign more energy to the interesting inputs.

As pointed out in Beacon [2], one limitation of this line of research is that it relies on heuristics to prioritize inputs and offers no guarantee that such prioritization could always improve the fuzzers' performance. For instance, AFLGo favored inputs with smaller distances. However, a few vulnerabilities are hidden deeply in the longer paths [35] and the inputs to trigger these vulnerabilities could not be favored. Hawkeye adopted other heuristics to mitigate this problem and focused on limited scenarios [1]. Another limitation of the existing distance-based approaches is that they usually could not improve fuzzing efficiency much. For instance, the most recent work WindRanger performed better than AFLGo by just around 44% [6].

2) Input Reachability Analysis: Identifying the unreachable inputs is another way to filter out uninteresting inputs. The inputs failing to reach the targets cannot trigger the vulnerabilities in the target code and thus could be safely identified as uninteresting ones. Recent works in this direction proposed two approaches.

Deep Learning. FuzzGuard [17] trained a deep learning model to identify and discard unreachable inputs. The key idea is to identify the pre-dominating nodes iteratively from the fuzzing targets, then filter out the inputs that could not pass through the pre-dominating nodes. FuzzGuard built a model to predict if the inputs could pass through the pre-dominating nodes by learning from previous executions. FuzzGuard was built upon AFLGo and achieved 5.40× speedup on average compared with AFLGo.

```
1  int main() {
2    int x,t=input();
3    ...
4    if(x<20) {
5      x=x+10;
6    }
7    if(t>20) {
8      t=t-10;
9    }
10   foo();
11   int y,z=input();
12   // assert(y<20&&z<40); // Beacon's precondition
13   ...
14   if(y<20) {
15     while(z<40) {
16       crash(x+z); // the fuzzing target
17       z++;
18     }
19   }
20   // assert(false); // Beacon's precondition
21   ... // other code
22   return 0;
23 }
```

Listing 1: A motivating example.

Path Pruning. Beacon adopted path pruning to early terminate execution on the unreachable paths [2]. It inserted checkpoints on variable-defining statements and branch statements. Specifically, Beacon performed interval analysis to infer the preconditions of reaching the target code and inserted assertions to ensure the preconditions were satisfied. It also terminated the executions on the branches that would never reach the targets. Beacon is considered as a state-of-the-art tool. It could be integrated with AFLGo and outperformed AFLGo by 11.50×.

Discarding unreachable inputs could effectively improve directed fuzzing performance. However, this approach considers only reachability. As a result, the directed fuzzers could waste much energy on exploring reachable yet irrelevant code.

Summary: Identifying the inputs interesting to reaching the targets is a promising direction in directed fuzzing research. Existing works identify interesting inputs based on distance metric or by discarding the unreachable inputs. Both approaches have non-trivial limitations—the first would not improve fuzzing efficiency much and could not guarantee the effectiveness, and the second considers only reachability and could waste lots of energy on exploring irrelevant code.

III. Problem Statement

In this section, we provide a motivating example to help understand relevant code (§III-A). We then discuss the limitations of prior directed fuzzers (§III-B). Finally, we present our research goals and the challenges in overcoming the limitations of the existing techniques (§III-C).

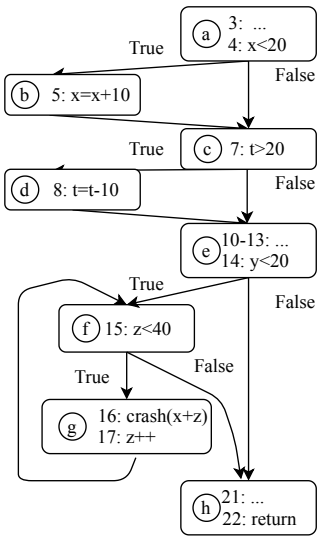

Fig. 1: Control-flow graph of the motivating example.

A. Relevant Code

Intuitively, *relevant code* is the code whose executions progress the directed fuzzing procedure for reaching the targets or exploiting the vulnerabilities. Since triggering a vulnerability requires satisfying certain control-flow and data-flow conditions, we define relevant code as code that determines if the target code can be reached (*i.e.*, control-flow conditions) or if the vulnerability can be exploited (*i.e.*, data-flow conditions). Note that this is different from analyzing whether the executions would approach the targets. For instance, if the executions approach the targets by executing the only successor *suc* of a reachable basic block *bb*, executing *suc* does not help further reach the targets as the likelihoods of reaching the targets from *bb* and *suc* are the same.

Take Listing 1 as an example, the fuzzing target is in line 16. The corresponding control-flow graph is depicted in Figure 1. Based on the definition, it is obvious that the unreachable code is always irrelevant to reaching the targets. Therefore, a recent work Beacon [2] inserts assertions in line 12 and line 20 to prevent the execution of unreachable code.

However, code reachable to the targets might also be irrelevant code as executing such code would not help further reach the targets or exploit the vulnerabilities. In Listing 1, only the code in lines 14-15 necessarily determines if the executions could reach the target. If we consider control-flow conditions, a lot of reachable code (*i.e.*, code in lines 2-10) is irrelevant as the executions could always reach line 11 by taking any paths. Furthermore, even that some reachable code in lines 11-13 has *indirect* control-flow dependency with the target, *i.e.*, they influence the conditional variables in lines 14-15, instrumenting it provides limited benefits and non-trivial side effects (more details are in §IV-B1).

We could not exclude all code blocks *irrelevant to reaching the targets* from the fuzzing scope as some of them might help the exploitation stage. For instance, executing line 5 would not affect whether the executions reach the target, but it changes the values of variables (*i.e.*, `x`) used in the target. We thus include such data-dependent code as relevant code.

B. Limitations of Existing Approaches

In this subsection, we illustrate that exploring irrelevant code would limit directed fuzzing efficiency. In particular, we show that existing tools are not efficient because they explore much irrelevant code. We present our idea using the motivating example in Listing 1.

Suppose the program takes an input *seed* and its execution path is ⓐ→ ⓒ→ ⓔ→ ⓗ. Fuzzers generate new inputs from *seed*, and *M* inputs among the new inputs uncover new code. Among the *M* inputs, the executions of *N* inputs uncover ⓕ or ⓑ. We consider the *N* inputs *relevant inputs* as their executions uncover relevant code, *i.e.*, ⓕ and ⓑ. The rest *M-N* inputs are *irrelevant inputs* as they uncover only irrelevant code. In particular, they make no progress in reaching the targets compared to *seed* as they all diverge from the targets at ⓔ. They also trigger no new critical operations (that change the value of `x` or `z`). Our experiments in §VI-C2 showed that the input queue could contain mainly irrelevant inputs because the majority of code in a program is irrelevant to a limited number of targets.

A directed fuzzer would suffer from low efficiency by exploring irrelevant code because of the following reasons. First, the mutations on irrelevant inputs would likely generate new irrelevant inputs that take the fuzzers to continue exploring irrelevant code. To efficiently *reach* the target code, we argue that a fuzzer should assign most energy to the inputs whose executions reach ⓕ, which is necessary for reaching ⓖ. However, we found that the existing directed fuzzers could unexpectedly assign more energy to irrelevant inputs whose executions always diverge at ⓔ. Even that the AFLGo-based tools allocate less energy to each irrelevant input as its distance is larger, they could still allocate much energy in total because the input queue contains primarily irrelevant inputs.[1] Beacon achieves a good performance by decreasing the number of irrelevant inputs. Specifically, it includes only the inputs uncovering reachable code in the input queue. Yet uncovering reachable code is not equivalent to progressing towards the targets as many inputs would uncover irrelevant (yet reachable) code blocks (*e.g.*, ⓓ). Similar to the inputs uncovering unreachable code, these inputs do not progress towards the targets, either. However, Beacon still favors them as their executions discover new code blocks.

Second, even the executions reached the target code (*e.g.*, ⓖ), a fuzzer would not be able to efficiently trigger the vulnerabilities if it extensively explored the irrelevant code. Specifically, a vulnerability could only be triggered when the program state (in the target code) satisfies certain conditions [36]. Yet exploring irrelevant code could not change the state in target code as irrelevant code has no data dependency with the target code. For instance, suppose $input_{rea}$ reaches ⓖ yet the

[1]The distance metric of AFLGo-based tools influences energy allocation up to 32×, yet the number of irrelevant inputs could be orders of magnitude larger than that of relevant inputs.

vulnerability is not triggered in current values of `x` and `z`, and the new inputs derived from $input_{rea}$ uncover code blocks in function `foo()` (line 10). Existing fuzzers take these inputs as interesting inputs as they could continuously uncover new code blocks in `foo()` even though they are not data dependent with ⓖ. However, such exploration could be useless but preempt the fuzzer's energy on mutating the relevant inputs (that uncover some critical code blocks such as ⓑ). Neither AFLGo-based tools nor Beacon could avoid such exploration because 1) the executions reach the target code (and thus are favored by AFLGo-based tools); and 2) the executions uncover reachable code blocks (and thus are favored by Beacon).

In summary, since existing directed fuzzers prioritize in mutating primarily irrelevant inputs thus explore mostly irrelevant code, their efficiency in either reaching the target code or triggering the vulnerabilities is not great.

C. Research Goals and Challenges

In this work, we aim to improve directed fuzzing efficiency by developing a better path exploration strategy that focuses on the interesting code relevant to the targets. Overly fuzzing irrelevant code would significantly limit fuzzing performance. Based on this observation, we aim to advance directed fuzzing by avoiding the exploration of irrelevant code.

We face two challenges in achieving our research goals. First, although we identify that the programs usually contain irrelevant code, there is still no formal definition of the (ir)relevant code in the context of directed fuzzing. Given the complex states of real-world programs, it is not trivial to automatically and precisely determine the code that helps the fuzzers reach the targets or trigger the vulnerabilities. Besides, how to avoid exploring irrelevant code is another challenge. Existing approaches like early termination do not work for the reachable yet irrelevant code. An intuitive strategy is to remove the irrelevant code such that the fuzzers never execute irrelevant code. However, it would not work well as removing irrelevant code might cause errors before the executions reach the targets. Even if the executions could normally reach the target code, the inputs might not be valid PoC exploits as they exploit different (*i.e.*, the simplified) programs. Therefore, it is necessary to propose other strategies to avoid exploring the irrelevant code.

IV. SELECTFUZZ

In this section, we present the design details of SELECTFUZZ, a directed fuzzer employing selective path exploration for efficient crash reproduction and vulnerability detection. It overcomes the limitations of the existing works by identifying and exploring only the code relevant to the fuzzing targets.

The overall architecture of SELECTFUZZ is depicted in Figure 2. It leverages the new distance metric to identify all the reachable code. The distance metric estimates the probability of reaching the target code for better input prioritization. It enables SELECTFUZZ to more efficiently approach the target code compared to the existing distance metrics. SELECTFUZZ further employs an inter-procedural control-flow and data-flow analysis to identify and selectively instrument only the relevant code for code coverage feedback. It assigns the mutation energy to an input based on its runtime relevant code coverage and input distance feedback. The selective path exploration allows SELECTFUZZ to collect high-quality runtime feedback from only the relevant code blocks and select better interesting inputs.

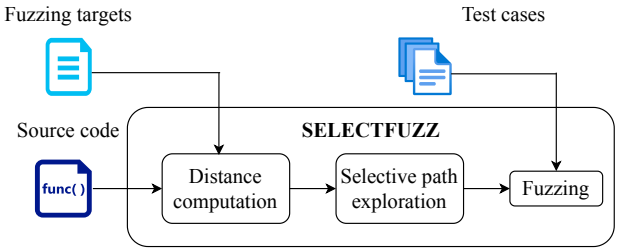

Fig. 2: Architecture of SELECTFUZZ.

A. Distance Metric

We develop a novel distance metric that estimates the *multi-path reaching "probability"* from a basic block to the target code. The *input distance* of an input is computed from the inter-procedural *block distances* to the target(s) of all the *relevant* basic blocks—including the ones in other functions—on the path it explores.

Compared to the existing distance metrics [1, 12, 16], our distance metric has the following advantages: 1) it comprehensively assesses the probability to reach the target code from a basic block by considering all the possible paths; 2) it measures the cross-function distance through a precise inter-procedural control-flow analysis and call target analysis. It allows SELECTFUZZ to better identify the inputs that are more likely to reach the targets.

1) Block Distance: The algorithm to compute the block distance is shown in Algorithm 1. It labels a basic block with three statuses—0: initial; 1: the distance is being computed; 2: the distance has been computed. The algorithm first computes the *reaching "probability"* P_b of a basic block b in function `cal_prob()`. P_b estimates the probability to reach the target location T from b. It then computes the inverse of the reaching probability as the block distance to the target, *i.e.*, $d_{bb}(b, T)$, in function `cal_dist()`.

The key of Algorithm 1 is function `cal_prob()`. If the basic block b is in the target location, its reaching probability is set to 100% (lines 25-26). Otherwise, it recursively computes the reaching probability from the successor blocks (lines 28-36). Specifically, the reaching probability of a block b is the average of the reaching probabilities of all its successor blocks. We do not consider the complexity of the conditional expressions and assume each branch of a conditional statement could be taken with the same probability. This simplification is needed as the real probability depends on the logical expressions and the corresponding inputs. Techniques like symbolic execution would help approximate the reaching probability better, yet

Algorithm 1 Block distance calculation.

Input : T, the target code locations;
Output : $dist$, a map to store the block distance.

$dist \leftarrow \{\}$
$prob \leftarrow \{\}$
$bb\_set \leftarrow build\_ICFG()$ // build the inter-procedural CFG and return all basic blocks
for b in bb_set **do**
    $b.status \leftarrow 0$
    $prob[b] \leftarrow 0$
end for
for b in bb_set **do**
    $dist[b] \leftarrow cal\_dist(b)$
end for
function cal_dist(BB b)
    $p = cal\_prob(b)$
    if $p == 0$ **then**
        return ∞
    else
        return $1/p$
    end if
end function
function cal_prob(BB b)
    if $b.status == 0$ **then**
        $b.status \leftarrow 1$
        if b in T **then**
            $prob[b] \leftarrow 1$
        else
            $sum \leftarrow 0$
            for $succ$ in $b.Successors$ **do**
                $prob[succ] \leftarrow cal\_prob(succ)$
                $sum \leftarrow sum + prob[succ]$
            end for
            $num \leftarrow b.getNumSuccessors()$
            if $num > 0$ **then**
                $prob[b] \leftarrow sum/num$
            end if
        end if
        $b.status \leftarrow 2$
    end if
    return $prob[b]$
end function

they still require the input distributions to make an accurate estimation. We leave this as future work. In the case of multiple fuzzing targets, the reaching probability of a block estimates the probability to reach any target blocks.

Programs usually contain loops and recursions. Since we cannot accurately infer the number of running iterations of loops/recursions in static distance analysis, we follow the common practice [37, 38] to unroll loop bodies and recursive functions once. For instance, we treat ⓕ in Figure 1 as an `if` statement and compute its reaching probability (and block distance) only once.

```
1  void foo(x,y) {
2    if(x>10) {
3      if(y>10)
4        y=y-10;
5      else
6        y=y+10;
7      crash();
8    }
9    return;
10 }
11
12 void bar(x,y) {
13   if(x>10)
14     if(y>10)
15       crash();
16   return;
17 }
```

Listing 2: An if-else statement and a nested if statement.

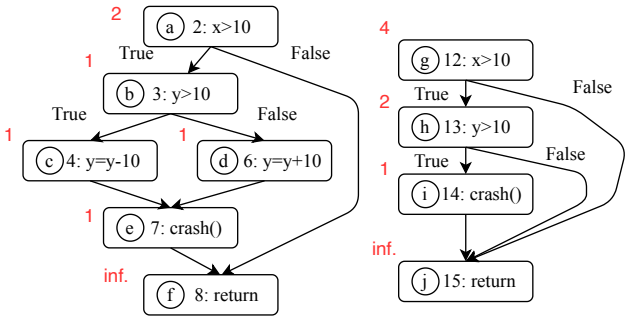

Fig. 3: Control flow graphs of the code in Listing 2. The red numbers are the block distances using our distance metric.

To compute the cross-function block distances, the entry block of a callee function is added as a successor of a call site. In case of indirect calls, we precisely infer the target functions and conservatively link the indirect call sites to the entry blocks of all the possible target functions (more details are in §V). This design choice allows SELECTFUZZ to further identify all relevant code (§IV-B1). However, it could also introduce some false positive paths to reach the target code. Nevertheless, the corresponding inputs would not be favored as the actually measured input distance would be quite large in the cases of false positives.

2) Input Distance: In our distance metric, the input distance of an input is the shortest block distance of all the basic blocks that are covered in its run. This is different from the prior works that compute the average block distance of the covered basic blocks as the input distance [1, 16]. The intuition behind this design choice is that the shortest distance could better reflect the progress of reaching the target code. In other words, it measures how close the execution is to the target.

3) An Example: We further use Listing 2 to help illustrate how our metric improves input prioritization. The corresponding control flow graphs are depicted in Figure 3. ⓐ has two successors, and one successor could always reach the target code. Therefore, we get $P_a = (P_b + P_f)/2 = 0.5$ and $d_{bb}(a, T) = 1/P_a = 2$; ⓗ also has two successors. We get $P_h = (P_i + P_j)/2 = 0.5$ and $d_{bb}(h, T) = 1/P_f = 2$, and similarly $d_{bb}(g, T) = 4$. Suppose we have two inputs *S1* and *S2*, and their execution traces are ⓐ→ ⓕ and ⓖ→ ⓙ (*i.e.*, `x<=10`), respectively.

Using the distance metric in AFLGo, we get $d_{input}(S1, T)$ = $3/1$ = 3 and $d_{input}(S2, T)$ = $2/1$ = 2. AFLGo would thus assign more energy to *S2* as $d_{input}(S2, T) < d_{input}(S1, T)$. However, by analyzing the control flow, we could infer *S1* is *more likely* to reach the target. Using our distance metric, we get $d_{input}(S1, T) = d_{bb}(a, T) = 2$; and $d_{input}(S2, T) = d_{bb}(g, T) = 4$. Consequently, SELECTFUZZ would prioritize in *S1* over *S2*.

B. Selective Path Exploration

SELECTFUZZ selectively instruments and explores only code that is relevant to the fuzzing targets. In the next, we first discuss how we identify the relevant code (§IV-B1), then introduce our input prioritization and power scheduling mechanisms (§IV-B2).

1) Relevant Code Identification: We identify the code with which the target code is control- or data-dependent as relevant code. Intuitively, code that can determine the control-flow or data-flow conditions for triggering the vulnerability is the relevant code. We define two classes of relevant code: *path-divergent* code and *data-dependent* code. The path-divergent code is in the last intersection block of a reachable path and an unreachable path. The data-dependent code influences the values of critical variables used in the targets; those variables can be assigned values of some constants or variables (direct dependency), which could further derive from other variables in other code blocks (indirect dependency). Note that the targets are included as relevant code and are considered as data-dependent code.

Path-divergent code provides important feedback with which SELECTFUZZ could distinguish the inputs that progress towards the targets. Specifically, by reaching a path-divergent code block, a positive feedback would be collected as it has a non-infinite block distance to the targets. SELECTFUZZ thus generates new inputs from the better seed inputs and keeps approaching the targets. It might also generate irrelevant inputs whose executions uncover only irrelevant code. However, it is unaware of irrelevant code as the code is not instrumented and provides neither coverage nor distance feedback. The irrelevant inputs are thus not favored; this prevents SELECTFUZZ from generating new irrelevant inputs. This is exactly what we optimize as such inputs provide feedback that drives the fuzzers towards different directions from the target code.

Note that we do not instrument the irrelevant code that could indirectly affect the control flow at the path-divergent code. For instance, in Listing 1, the path-divergent code in line 14 or line 15 might be data-dependent on some irrelevant code block(s) omitted in line 13. We trade off instrumenting such code for a much smaller exploration space. We notice that the reachability towards the targets is mainly influenced by the path-divergent code rather than the other code. Specifically, the program would keep executing on the reachable path (after it reaches one path-divergent block) until it reaches the next path-divergent block. The indirect control-dependent code blocks (*i.e.*, those influencing variables in conditions) on the reachable path would be visited when the fuzzer tries to reach the path-divergent code block regardless if we instrument them or not. Instrumenting the other code on the reachable path does not provide additional helpful feedback but would increase the runtime overhead. Our observation is also consistent with the results in a recent work—WindRanger [6], which demonstrates that only the "deviation code blocks" are important in reachability analysis.

The data-dependent code might not help reach the target code, but can help with the exploitation. Since some vulnerabilities may only be triggered when the critical variables in targets are in certain values [36], exploring the data-dependent code could help the exploitation stage of SELECTFUZZ.

SELECTFUZZ currently does not—but can—prune the unreachable paths like Beacon [2]. As path pruning saves unnecessary execution time on unreachable paths, we could integrate SELECTFUZZ with this approach to further improve the fuzzing efficiency. We will present the evaluation results and more discussion about it in §VI-A3.

Back to the example in Figure 1, according to our definition, the relevant code includes ⓑ, ⓔ, ⓕ, ⓖ. Specifically, ⓔ and ⓕ are the path-divergent code; ⓑ and ⓖ are the data-dependent code. Compared to Beacon that explores the state space of all code blocks except ⓗ, SELECTFUZZ refines the fuzzing scope to only four code blocks. This allows it to more efficiently trigger the potential erroneous states in the target code.

2) Input Prioritization and Power Scheduling: As we use a different instrumentation strategy for directed fuzzing, we accordingly change the input prioritization and power scheduling mechanisms in standard fuzzers. In this subsection, we introduce SELECTFUZZ's input prioritization and power scheduling mechanisms.

Input Prioritization. SELECTFUZZ selects the relevant inputs that cover new edges between *relevant basic blocks* or increase the hit count of an existing edge to a new scale. SELECTFUZZ might not always find relevant inputs. For instance, it might fail to generate an input satisfying a critical condition. In cases where no relevant inputs are found, SELECTFUZZ would select and mutate the irrelevant inputs with smaller input distances.

Power Scheduling. SELECTFUZZ adopts the annealing-based power scheduling method in AFLGo [16]. In particular, it gradually assigns more energy to the inputs that are closer to the target locations. Some related works proposed different power scheduling algorithms (*e.g.*, the adaptive power scheduling in [27]). We could but do not adopt other power scheduling algorithms for a fair comparison with AFLGo (§VI-A3). SELECTFUZZ does not include non-instrumented code when computing input distance, which might affect power scheduling. We next show that the selective instrumentation does not affect it.

According to §IV-B1, if a basic block $irbb$ is not a relevant basic block, it either 1) has a single successor or 2) all its successors can or cannot reach the targets. In the first case, $d_{bb}(irbb, T)$ is equal to the distance of its only successor, thus the input distance does not change regardless if $irbb$ is instrumented or not. In the second case, assuming all

successors have the same block distance, $d_{bb}(irbb, T)$ would be the same as its successors', so the input distance does not change with respect to $irbb$. The successors of $irbb$ could also have different distances. Assume that successor $succ1$ has the shortest distance that is smaller than $d_{bb}(irbb, T)$ and successor $succ2$ has the largest distance that is larger than $d_{bb}(irbb, T)$. Then the distances of the inputs visiting $succ1$ would not be affected (*i.e.*, they are always the shortest distance $d_{bb}(succ1, T)$). The distances of the inputs visiting $succ2$ would be $d_{bb}(succ2, T)$ if we do not instrument $irbb$, or $d_{bb}(irbb, T)$ if we instrument $irbb$ (because $d_{bb}(irbb, T) < d_{bb}(succ2, T)$). However, since $d_{bb}(succ1, T) < d_{bb}(irbb, T) < d_{bb}(succ2, T)$, the inputs visiting $succ2$ would not be favored compared to the inputs visiting $succ1$ regardless of whether $irbb$ is instrumented. Therefore, the power scheduling works as intended without instrumenting $irbb$.

V. Implementation

We implemented a prototype of SelectFuzz with around 2,100 lines of C++ code, including ~1,400 LoC for distance measurement and ~700 LoC for selective instrumentation. Distance measurement and compile-time instrumentation were both implemented as compiler passes. SelectFuzz supports analyzing applications with LLVM bytecode. The source code of our prototype will be available at https://github.com/cuhk-seclab/SelectFuzz. Next, we discuss a few important implementation details.

Call Graph. SelectFuzz builds call graphs when it computes the cross-function distance. It infers the target functions in indirect calls using Andersen's points-to analysis [39]. Additionally, we optimize the imprecise call graph with argument-type-based pruning and address-taken-based pruning [40]. According to [40], the optimizations could reduce the number of call edges by 70% while retaining correct call edges. Since the current implementation[2] supports only C programs, we extend it to C++ programs by supporting polymorphism features. We infer the target functions of method calls (*e.g.*, `Obj.func()`) by analyzing the receiver class types. We then analyze the polymorphism and find the target functions in the correct classes.

Inter-procedural Data-flow Analysis. SelectFuzz performs backward inter-procedural data-flow analysis starting from the target code to capture data-dependent code. It first identifies the critical variables used in the targets and then finds other code that might influence the critical variables. To support the inter-procedural analysis, SelectFuzz analyzes the callee function of each call site, and identifies the arguments/parameters that the critical variables are data dependent on. It then analyzes the data-flow relationship on callee function to infer if its return values have data dependency with the critical variables. Finally, it propagates the return values of callee function to the correct call site in a context-sensitive manner like in [40].

Alias Analysis. Our data-flow analysis incorporates the conservative points-to analysis in [39, 40] to handle pointer aliasing. Because of the conservative points-to analysis, SelectFuzz has false positives in identifying the aliases thus also the data-dependent code. For example, it might incorrectly determine that two pointers point to the same memory location and consider that they have data dependencies. Advancing points-to analysis is an open challenge and orthogonal to this work. Since our focus is to dynamically trigger the vulnerabilities, similar to other works [40, 41], we do not particularly address the false positives in existing static analysis tools. We will discuss the impacts of the false positives in §VII.

Execution Timeout. SelectFuzz might insert inputs triggering timeout into the input queue and assign certain energies to them. Existing fuzzers discard inputs if their executions trigger timeout for time efficiency. However, the inputs triggering timeout might uncover new relevant basic blocks and thus move the fuzzing progress forward. Because only a small proportion of inputs are relevant to the target code and make a progress, such inputs are always retained for future mutations. However, inputs triggering timeout will be assigned with less energy compared with the normal inputs with the same input distance.

VI. Evaluation

In this section, we evaluate SelectFuzz using real-world vulnerabilities and answer the following questions:

- **RQ1** How effective is SelectFuzz in triggering known vulnerabilities?
- **RQ2** How does each component of SelectFuzz contribute to its performance?
- **RQ3** Which factors affect SelectFuzz's effectiveness?
- **RQ4** How does SelectFuzz perform in a standard fuzzing benchmark?
- **RQ5** Can SelectFuzz detect new vulnerabilities in real-world programs?

A. Triggering Known Vulnerabilities (RQ1)

1) Settings: We choose a set of known vulnerabilities evaluated in prior works [1, 2, 16] as the evaluation dataset. These vulnerabilities reside in real-world programs that handle different file formats, including PDF, executable, XML, *etc.* The vulnerabilities and the corresponding programs are listed in the second and third columns of Table I, respectively.

Directed fuzzers require setting initial seeds and specifying the target code. In our experiment, we use the seeds provided by AFLGo in the first eleven cases (*i.e.*, No. 1-11) and obtain seeds from the corresponding code repositories in the rest cases (*i.e.*, No. 12-18).[3] We search and select the final crash point as the fuzzing target for reproducing a crash. For a fair comparison, we apply the same seeds and fuzzing target per vulnerability across different directed fuzzers. Similar to Beacon [2], we conduct the experiment 5 times and set a time budget of 120 hours. All experiments are conducted in Docker running on a 64-bit Ubuntu machine with 8 CPU cores (Intel Xeon(R) CPU W-2123 @ 3.60GHz) and 16 GB of memory.

[2]https://github.com/shamedgh/temporal-specialization

[3]Cases No. 12-18 were evaluated in other works but their initial seeds have not been released.

Table I: The evaluation dataset and the average crash exposure time of AFLGo, SELECTFUZZ, Beacon†, and SELECTFUZZ*. BB_{total}, BB_{rec}, and BB_{rel} denote the total number of basic blocks, the number of reachable basic blocks, and the number of relevant basic blocks in the programs, respectively. Beacon† denotes our naive path pruning implementation of Beacon; SELECTFUZZ* denotes the integration of SELECTFUZZ and Beacon†. We obtained Beacon's speedup over AFLGo from its paper [2].

No.	CVE-ID	Program	Vuln. Code	BB_{total}	BB_{rec}	BB_{rel}	AFLGo	SELECTFUZZ	Beacon†	SELECTFUZZ*	Speedup	
											SELECTFUZZ	Beacon
1	2016-9827	swftophp-0.4.7	outputtxt.c:144	4,114	717	192	1.07 h	0.25 h	0.46 h	0.22 h	**4.28**	3.90
2	2017-7578	swftophp-0.4.7	parser.c:68	3,788	323	116	2.41 h	0.77 h	0.83 h	0.70 h	3.13	**8.10**
3	2018-8807	swftophp-0.4.8	decompile.c:349	3,798	626	298	6.23 h	0.32 h	3.74 h	0.27 h	**19.47**	5.67
4	2018-8962	swftophp-0.4.8	decompile.c:398	3,798	516	198	29.70 h	0.64 h	4.81 h	0.33 h	**46.41**	18.37
5	2017-11728	swftophp-0.4.8	decompile.c:868	3,798	484	221	120 h	42.17 h	32.10 h	19.87 h	2.85	**10.76**
6	2018-20427	swftophp-0.4.8	decompile.c:425	3,798	91	56	120 h	15.26 h	12.78 h	5.14 h	7.86	**35.10**
7	2017-8846	lrzip-0.631	stream.c:1756	9,454	511	325	120 h	120 h	120 h	120 h	1	N.A.
8	2016-4491	cxxfilt-2.26	cp-demangle.c:4318	42,150	414	163	10.73 h	3.38 h	4.13 h	3.09 h	3.17	**3.87**
9	2017-9047	xmllint-20904	valid.c:1279	69,696	18,032	1,094	120 h	10.28 h	42.30 h	9.92 h	**11.67**	6.66
10	2017-9048	xmllint-20904	valid.c:1323	69,696	18,046	1,021	120 h	24.87 h	45.57 h	16.47 h	4.83	**6.01**
11	2017-9049	xmllint-20902	parser.c:3406	69,853	20,128	4,540	120 h	104.39 h	71.83 h	33.04 h	1.15	**3.62**
12	2017-8392	objdump-2.28	dwarf2.c:4212	60,482	463	331	1.44 h	0.51 h	0.40 h	0.24 h	2.82	N.A.
13	2017-8396	objdump-2.28	libbfd.c:615	60,518	13,123	3,135	1.98 h	0.53 h	0.65 h	0.44 h	3.74	N.A.
14	2017-8397	objdump-2.28	reloc.c:885	60,518	2,800	697	120 h	2.73 h	43.20 h	2.03 h	43.96	N.A.
15	2017-14940	objdump-2.28	parser.c:3406	60,482	11,601	2,418	2.63 h	1.71 h	2.46 h	1.31 h	1.53	N.A.
16	2019-10872	libpoppler.so.87	Splash.cc:5872	84,662	2,028	91	120 h	5.02 h	59.30 h	4.84 h	23.90	N.A.
17	2019-10873	libpoppler.so.86	SplashXPathScanner.cc:458	84,529	3,942	281	41.52 h	4.13 h	21.25 h	4.03 h	9.95	N.A.
18	2019-14494	libpoppler.so.89	SplashOutputDev.cc:4584	84,925	2,231	111	120 h	120 h	120 h	118.89 h	1	N.A.

2) Performance of SELECTFUZZ*:* We measured the time used by SELECTFUZZ to reproduce the crashes. We present the evaluation results in the ninth column of Table I. In general, SELECTFUZZ successfully reproduced crashes in 16 out of the 18 cases. It failed to reproduce 2 crashes (*i.e.*, No. 7 and No. 18) within the time budget, while other fuzzers could not trigger them, either. The authors of Beacon claimed to have triggered the two cases [2]. The different results might be caused by the different seed inputs used in our experiment and theirs. The seed corpus could significantly influence the fuzzing efficiency [42]. To be consistent with our seed selection strategy, however, we did not separately use other seed inputs for the two cases. We will discuss it in details in §VI-C2.

3) Comparison with Prior Works: We compare SELECTFUZZ with the baseline tool—AFLGo [16]—and a state-of-the-art tool—Beacon [2]. We do not include other advanced directed fuzzers such as Hawkeye [1] into our evaluation as Beacon outperformed them in its evaluation. We do not include directed fuzzers attempting to automatically specify the target code [3, 27, 30] either as they have a different focus from SELECTFUZZ. Note that Beacon is not open-sourced and we used its binary in the Docker image provided by the authors [43] in our evaluation.

Comparison with AFLGo. The evaluation results of AFLGo are listed in the eighth column in Table I. AFLGo reproduced crashes in 9 out of the 18 cases and failed in 9 cases. SELECTFUZZ outperformed AFLGo by successfully reproducing 7 more crashes. We carefully compared the time used for those successful cases. SELECTFUZZ significantly improved AFLGo with an over 10× speedup in 5 (27.8%) cases and up to 46.41× speedup. The results demonstrate that our selective path exploration strategy is highly effective in improving directed fuzzing efficiency. We further provide the characterization of SELECTFUZZ's effectiveness in §VI-C.

Comparison with Beacon. We were unable to fairly evaluate Beacon in our experiments and had to reuse the results reported by the authors [2]. We used the Beacon binary in the Docker image the authors released [43]. We inferred that this binary was the implementation of Beacon above AFL (*i.e.*, Beacon+AFL), though the authors stated that they used AFLGo as their primary fuzzing engine [2]. The reason is twofold. First, our experiment results were inconsistent with the results the authors presented in their evaluation using Beacon+AFLGo [2]. Also, our binary analysis (*e.g.*, reverse engineering and binary diffing [44]) revealed that the provided fuzzing engine was almost identical to the standard AFL—it lacked a few key components (*e.g.*, distance metrics) and the unique fuzzing options (*e.g.*, -z) of AFLGo. The performance of Beacon+AFLGo should be better than that of Beacon+AFL according to their paper [2], For a fair and desired comparison, we should compare SELECTFUZZ with Beacon+AFLGo, which is not publicly available. We could thus only reuse the results reported in the paper [2] to represent the speedup of Beacon+AFLGo. In the following, we use Beacon and Beacon+AFLGo interchangeably as we mainly compare SELECTFUZZ with Beacon+AFLGo.

The authors of Beacon did not publish their seed corpus while the selection of seed inputs is critical for fuzzing. However, the authors mentioned that they would use the seeds if provided by AFLGo [2]. Therefore, we were able to compare the performance of SELECTFUZZ with Beacon in the first eleven vulnerabilities (*i.e.*, No. 1-11), which were evaluated by both AFLGo and Beacon. Note that we excluded case No. 7, for which we measured quite different crash exposure time of AFLGo, which led us to infer that Beacon used different seeds in that case.

We list the speedup of Beacon upon AFLGo in the last column of Table I. Overall, the results indicate that selective path exploration and path pruning could both improve directed fuzzing efficiency, but the improvements differ across cases. As we discussed in §IV-B1, SELECTFUZZ avoided exploring much irrelevant reachable code that Beacon explored. It also had less instrumentation overhead compared to Beacon (§VI-C). Therefore, SELECTFUZZ performed better than Beacon in 4 (out of 10) cases even without adopting path pruning. It also

achieved significantly higher efficiency than Beacon in some cases such as No. 3 and 4. Beacon performed better than SELECTFUZZ in several cases. The reason is that SELECTFUZZ *executed* some unreachable code, which could be addressed by path pruning.

Therefore, to measure the exact benefit of our selective path exploration strategy, we have to integrate SELECTFUZZ with Beacon which is not open-sourced. To this end, we implemented a naive prototype of Beacon (we name it Beacon†) to early terminate unreachable paths and further integrated SELECTFUZZ with it. Beacon† differs from Beacon in the aspect that Beacon† prunes only the branches that could never reach the targets. We name the integration of SELECTFUZZ and Beacon† as SELECTFUZZ*.

We list the crash exposure time of Beacon† and SELECTFUZZ* in the tenth and eleventh columns of Table I, respectively. The evaluation results indicated that SELECTFUZZ* outperformed Beacon† in all cases, achieving a speedup of 5.23$\times$ on average. This suggested that the selective path exploration and the path pruning approaches are well complementary.

Finally, we directly compare SELECTFUZZ* to Beacon [2] using same seeds in the above 10 cases (No. 1-6 and 8-11). We found that SELECTFUZZ* performed better than Beacon in 6 cases, except for No. 2, 5, 6, and 8. Based on our experiments, we found that path pruning was very effective in the four cases. Therefore, by terminating more unreachable code paths, Beacon could further significantly improve the fuzzing efficiency. Nevertheless, selective path exploration still benefited the path pruning approach as SELECTFUZZ* improved Beacon† by 15.7%, 61.6%, 59.8%, and 25.2% in No. 2, No. 5, No. 6, and No. 8, respectively. We are confident that SELECTFUZZ* would also perform better than Beacon in the four cases if it had terminated all the unreachable paths identified by Beacon.

B. Ablation Study (RQ2)

As SELECTFUZZ improves directed fuzzing from two aspects—distance metric and selective path exploration, we perform an ablation study to evaluate the effectiveness of each component in this subsection. To evaluate our distance metric, we modify AFLGo by replacing its distance metric with ours; we represent this setup as AFLGo+Distance Metric. To evaluate selective path exploration, we instrument only relevant code but compute input distance using the distance metric of AFLGo; we represent this setup as AFLGo+Selective Path Exploration. The results are shown in Figure 4. Specifically, the blue, red, yellow, and black bars denote the crash exposure time using AFLGo, AFLGo+Distance Metric, AFLGo+Selective Path Exploration, and SELECTFUZZ, respectively.

We first discuss the effectiveness of our distance metric. Overall, our distance metric improved AFLGo by 39% on average. The results suggested that prioritizing inputs based on reaching probability (*i.e.*, our distance metric) is more efficient than always favoring the shortest code paths (*i.e.*, AFLGo's approach). AFLGo performed slightly better than our distance metric in 3 out of 18 cases (*i.e.*, No. 1, No. 12, and No. 13), although the exposure time in the 3 cases was trivial. Besides, our distance metric was quite effective in a few cases such as No. 14 and No. 16. We found that AFLGo did not perform well in those cases because it lacked support for a precise inter-procedural analysis. For instance, as *objdump* (No. 14) and *poppler* (No. 16) extensively use indirect calls, AFLGo misidentified many unreachable code blocks and that misled its input prioritization. SELECTFUZZ precisely resolved function calls when computing the input distance, thus addressing this problem.

We next study the effectiveness of selective path exploration. First, we found that selective path exploration greatly improved the fuzzing efficiency. On average, it enabled AFLGo to achieve an average 6.68$\times$ speedup. That substantiated our observation in §III-A that existing directed fuzzers would waste much fuzzing energy on tracing code coverage of irrelevant code. Besides, SELECTFUZZ performed better than AFLGo+Selective Path Exploration in most (17 out of 18) cases. This demonstrated the importance of integrating selective path exploration with our distance metric, which could further help select better inputs (from relevant inputs) to reach the targets and trigger the vulnerabilities.

C. Understanding Performance Boost (RQ3)

In this subsection, we discuss a few important factors that affect the performance of SELECTFUZZ and other directed fuzzers. In particular, we show that the instrumentation overhead, the selection of seed inputs, and the path constraints to target code could largely affect the crash exposure time.

1) Instrumentation Overhead: Instrumentation can introduce runtime overhead that slows down the instrumented program, thus affecting fuzzing efficiency. SELECTFUZZ identifies and selectively instruments only the relevant basic blocks therefore introduces low runtime overhead. To demonstrate this, we investigate the proportion of relevant basic blocks in real-world programs. The number of total basic blocks, reachable basic blocks, and relevant basic blocks are listed in the fifth, sixth, and seventh columns in Table I, respectively. Overall, SELECTFUZZ identified 15,288 basic blocks as relevant basic blocks (1.96% of the total basic blocks or 12.33% of the reachable basic blocks). In other words, SELECTFUZZ instruments significantly less code than AFLGo-based directed fuzzers.

However, not all instrumentation code would be executed in fuzzing. To understand the practical benefit of selective instrumentation, we measure the performance of the vanilla binaries and the binaries instrumented by SELECTFUZZ and AFLGo. We found that the run time of the binaries instrumented by SELECTFUZZ was almost the same as the vanilla binaries. In comparison, AFLGo's instrumentation made the programs 57% slower as the instrumented code is triggered upon executing each basic block.

2) Exploring Relevant Code: The effectiveness of SELECTFUZZ is largely affected by the proportion (*i.e.*, P_{ir}) of the triggered irrelevant code blocks among all triggered code blocks. Specifically, P_{ir} represents how much irrelevant code prior works would explore and SELECTFUZZ would exclude.

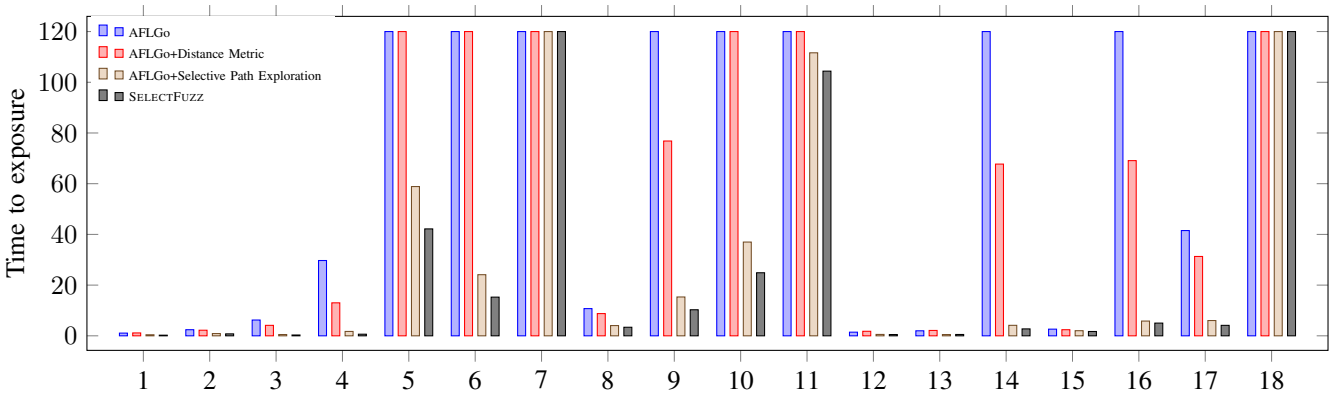

Fig. 4: The component-wise effectiveness of SELECTFUZZ. The y-axis is the crash exposure time.

We observed that P_{ir} was usually very large and that resulted in low efficiency in existing directed fuzzers. This could be explained by the results that only 1.96% / 12.33% of total / reachable code blocks are relevant. In the following, we further discuss a few factors affecting P_{ir} to better understand the performance boost.

Seed Inputs. Like in other fuzzing works [2, 16, 18], the seed inputs greatly affect the code triggered during fuzzing as all test inputs are derived from the seed inputs. A bad seed input might trigger much irrelevant code that existing directed fuzzers would explore but SELECTFUZZ can exclude. For example, the seed of No. 16 could already reach the target code location. However, it was a large PDF file (over 55KB) whose execution visited many irrelevant basic blocks. Because of the huge exploration state space, AFLGo failed to trigger the crash within 120 hours. SELECTFUZZ refined the fuzzing scope to the relevant basic blocks (<0.01% of the total basic blocks of the test program), and it reproduced this crash using about 5 hours. To understand the impacts of seeds, we further removed some unimportant input bytes from the seed of No. 16 and obtained a new PDF file with around 40 KB. Note that the new seed retained the critical input bytes (specifically, the *form* structures) that were relevant to triggering No. 16. The results showed that with the new seed, AFLGo triggered the same crash using 71.3 hours and SELECTFUZZ used around 4.3 hours. Crafting good seed inputs could thus improve fuzzing efficiency.

Complexity of Path Constraints. The complexity of the constraints on the paths to the target code also affects P_{ir}. Specifically, the inputs would trigger mostly irrelevant code blocks if most executions fail to satisfy the path constraints for reaching the target code. For example, we applied the same seed input (*i.e.*, a 3.2K ELF file) to test *objdump* in cases No. 12-15. AFLGo could reproduce 3 of them but failed for CVE-2017-8397 (No. 14). We found that compared to the other three cases, AFLGo could hardly generate inputs satisfying the constraints on the paths to the target code in this case. However, many inputs *unavoidably* uncovered other irrelevant code and its feedback drove AFLGo to explore other "easier" code paths. Different from AFLGo, SELECTFUZZ could efficiently distinguish a few inputs that progress towards the target code. Once some inputs solved a complex path constraint towards the target code, only they would be selected for further mutation. The other unsatisfying inputs would be discarded immediately even they would uncover other code blocks.

D. Benchmarking (RQ4)

We further evaluate SELECTFUZZ on a standard fuzzing benchmark to better understand its performance in finding different types of vulnerabilities.

1) Settings: We compare SELECTFUZZ with AFLGo [16] (as baseline), Beacon [43], Angora [45], ParmeSan [27], and AFLChurn [3] using the Google Fuzzer Test Suite (GFTS). Compared to §VI-A3, we include three more fuzzers (*i.e.*, [45], [27], and [3]) that specialize in generating high-quality inputs. This allows us to assess the strengths and limitations of different techniques (*e.g.*, path pruning, byte-level taint tracking, and selective path exploration). We do not include other fuzzers because they are not open-sourced (*e.g.*, [46] and [47]) or their performance (*e.g.*, [48]) was worse than Angora on GFTS.

We used the initial seeds provided in GFTS when available. We set a file containing "hi" as the initial seed for a few programs that the seeds are unavailable. Similar to §VI-A, we set the final crashing point as the target for each vulnerability. We also carefully configured the fuzzing options (*e.g.*, `-x` in testing *libxml*), which could have great impacts on fuzzing performance according to our experiments. Finally, we applied the same settings to run each fuzzer, and conducted each experiment 5 times with a time budget of 24 hours.

We followed [3, 49] to set up the fuzzers. Angora's compiler does not work for *boringssl* and we fixed it following [50]. ParmeSan [27] and AFLChurn [3] have an additional target acquisition step that identifies other target sites. We skipped it in our experiment and configured the final crashing point as the target to provide a fair comparison with other fuzzers.

2) Vulnerability Exposure: Table II shows the average time-to-exposure (TTE) of vulnerabilities of the evaluated fuzzers. Overall, SELECTFUZZ, Beacon, Parmesan, and AFLChurn performed the best in 8, 4, 4, and 1 cases, respectively. Moreover, SELECTFUZZ achieved good performance in triggering some complex cases. For instance, it used about 8 hours (the shortest TTE) in reproducing the specified crash in *libarchive* yet other

Table II: The average crash exposure time and the statistical test p values of the evaluated tools for each vulnerability in the Google Fuzz Test Suite. † Program *lcms* has two vulnerabilities that crash at the same code location.

Program	Vuln. Code	AFLGo		Beacon		ParmeSan		Angora		AFLChurn		SELECTFUZZ	
		TTE	*p*	TTE	*p*	TTE	*p*	TTE	*p*	TTE	*p*	TTE	*p*
boringssl	asn1_lib.c:459	T.O.	-	T.O.	-	**19.92 h**	0.043	T.O.	-	T.O.	-	T.O.	-
c-ares	ares_create_query.c:196	<0.01 h	0.011	<0.01 h	0.002	<0.01 h	0.008	<0.01 h	0.075	<0.01 h	0.006	<0.01 h	0.002
guetzli	output_image.cc:398	0.58 h	0.006	0.50 h	0.003	0.17 h	0.006	0.62 h	0.028	0.47 h	0.007	**0.09 h**	0.004
harfbuzz	hb-buffer.cc:419	T.O.	-	22.11 h	0.008	**16.08 h**	0.004	T.O.	-	T.O.	-	20.40 h	0.015
json	fuzzer-parse_json.cpp:50	0.07 h	0.006	**0.03 h**	0.000	0.04 h	0.002	0.07 h	0.006	0.06 h	0.005	0.05 h	0.001
lcms	cmsintrp.c:642	8.38 h	0.046	**4.97 h**	0.001	8.18 h	0.006	22.90 h	0.107	7.42 h	0.006	6.23 h	0.004
lcms	cmsintrp.c:642†	6.90 h	0.005	5.77 h	0.006	5.58 h	0.018	8.86 h	0.079	7.32 h	0.006	**3.52 h**	0.004
libarchive	archive_read_support_format_warc.c:537	T.O.	-	T.O.	-	13.04 h	0.037	18.06 h	0.219	T.O.	-	**8.12 h**	0.034
libssh	messages.c:1001	0.21 h	0.004	0.52 h	0.132	0.22 h	0.008	0.20 h	0.005	0.12 h	0.004	**0.09 h**	0.002
libxml2	parser.c:10666	0.09 h	0.006	0.02 h	0.002	**<0.01 h**	0.000	0.42 h	0.002	**<0.01 h**	0.001	**<0.01 h**	0.000
libxml2	dict.c:489	0.33 h	0.005	0.26 h	0.004	**0.08 h**	0.004	0.49 h	0.007	0.19 h	0.001	0.10 h	0.002
openssl-1.0.1f	t1_lib.c:2586	<0.01 h	0.001	<0.01 h	0.000	<0.01 h	0.000	<0.01 h	0.001	<0.01 h	0.002	<0.01 h	0.000
openssl-1.0.2d	target.cc:145	0.07 h	0.002	**0.02 h**	0.000	0.05 h	0.006	0.12 h	0.018	0.08 h	0.006	0.05 h	0.000
pcre	pcre2_match.c:1426	0.62 h	0.005	0.56 h	0.002	0.60 h	0.003	1.34 h	0.008	0.53 h	0.006	**0.40 h**	0.006
re2	nfa.cc:532	22.43 h	0.036	10.90 h	0.009	13.12 h	0.004	T.O.	-	13.46 h	0.018	**9.57 h**	0.011
vorbis	codebook.c:407	T.O.	-	**21.55 h**	0.040	T.O.	-	T.O.	-	T.O.	-	T.O.	-
woff	woff2_dec.cc:1274	6.47 h	0.006	6.04 h	0.003	4.93 h	0.001	12.57 h	0.028	6.32 h	0.006	**4.56 h**	0.004

fuzzers (*i.e.*, AFLGo, Beacon, and AFLChurn) failed to trigger it within 24 hours.

We also employed a Mann-Whitney U test [51] on the time-to-exposure to measure the statistical significance of our experiment results. It can be seen that in nearly all cases, the results are significant with $p < 0.05$. Therefore, the influence of random variations was limited. Compared to Angora, directed fuzzers usually had a lower p-value (*i.e.*, fewer random variations). Since directed fuzzers prioritize in reaching some specified code locations, intuitively, this reduces the randomness in exposing the vulnerabilities.

We further study why other fuzzers performed better than SELECTFUZZ in the 9 cases. One main reason is that SELECTFUZZ could not efficiently *generate* the high-quality inputs to satisfy some complex path constraints. It failed to trigger the crash in *boringssl* because of this. Incorporating other advanced input generation techniques (*e.g.*, byte-level taint tracking) could address this problem. Indeed, this vulnerability was only triggered by Parmesan, a directed fuzzer utilizing taint-guided input mutation. Besides, the low fuzzing throughput also hinders SELECTFUZZ's efficiency in triggering some crashes. For instance, when reproducing the crash in *vorbis*, it executed only a small number of (*e.g.*, usually less than 10) inputs per second. Beacon was the only fuzzer triggering that crash as it greatly improved the execution speed (*i.e.*, hundreds of inputs per second) with its path pruning technique.

The benchmarking results suggest that our techniques help improve directed fuzzing performance and can be combined with the existing techniques for achieving potentially better performance.

E. Detecting New Vulnerabilities (RQ5)

In this subsection, we apply SELECTFUZZ to detect new vulnerabilities. We include the applications evaluated in recent fuzzing works [2, 24, 28] as our testing objects and compile these programs using default configurations. Since directed fuzzers require specifying fuzzing targets, we investigate the recent vulnerabilities in these applications from the CVE database [52] and use their root cause locations as our fuzzing targets. The rationale is that these erroneous locations sometimes contain more than one vulnerability [28]. By setting them as the fuzzing targets, we are likely to find new vulnerabilities or reveal incomplete patches.

Table III: New vulnerabilities detected by SELECTFUZZ.

Project	Root Cause	Vul. Type	Status	CVE
poppler	Object.h:435	abort	patched	issue-1274
poppler	XRef.cc:1388	abort	patched	issue-1278
poppler	Object.h:435	abort	patched	issue-1276
poppler	PDFDoc.cc:1755	abort	patched	issue-1282
poppler	Object.h:445	abort	patched	issue-1289
poppler	pdftocairo.cc:731	assertion	confirmed	issue-1287
libjpeg	iostream.cpp:543	infinite loop	patched	2022-37768
libjpeg	losslessscan.cpp:374	seg. fault	patched	2022-37769
libjpeg	linemerger.cpp:262	seg. fault	patched	2022-37770
tcpreplay	get.c:344	heap overflow	patched	2022-37048
tcpreplay	get.c:150	heap overflow	patched	2022-37049
tcpreplay	get.c:713	heap overflow	patched	2022-37047
libtiff	tif_jpeg.c:962	assertion	confirmed	issue-445
libming	parser.c:2431	memory leak	-	issue-239

SELECTFUZZ identified 14 new vulnerabilities with a time budget of 72 hours. The results in Table III also show that it was able to identify various types of vulnerabilities, including heap overflow, segmentation fault, memory leak, *etc.* Moreover, SELECTFUZZ found new vulnerabilities in complex software like *poppler* and *libjpeg*. We responsibly reported all the new vulnerabilities to the relevant developers. At the time of writing, 11 vulnerabilities have been patched and 6 have been assigned with new CVE IDs.

We further provide an example to help understand the newly detected vulnerabilities. In CVE-2018-20662, the *poppler* program invoked `getDict()` (which is defined at `poppler/Object.h:435`) without checking the object type [53]. This vulnerability was fixed by adding a check of `isDict()` before the corresponding call site (*i.e.*, `utils/pdfunite.cc:172`) of `getDict()`. However, the developers called `getDict()` in other locations without applying the same check because those call sites were considered as not vulnerable. We set `poppler/Object.h:435` as the target, and SELECTFUZZ discovered two previously unknown vulnerable program paths that called `getDict()` without type checking. The vulnerabilities have high impacts because *poppler* is commonly

used in Linux systems[4]. After our report, the developers fixed them in two days by adding the `isDict()` check before the two newly found vulnerable call sites of `getDict()` (*i.e.*, `poppler/PDFDoc.cc:889` and `utils/pdfunite.cc:200`). Note that although *poppler* has been continuously fuzzed by the OSS-Fuzz project [54] and other fuzzers, the vulnerabilities had been hidden in the project for 3 years until SELECTFUZZ discovered it.

VII. DISCUSSION

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

False Positives in Relevant Code Identification. SELECTFUZZ could overestimate the relevant code. First, since SELECTFUZZ over-approximates the call relationships and also performs a conservative alias analysis in data-flow analysis, it has false positives in identifying both path-divergent code and data-dependent code. Second, SELECTFUZZ considers all variables used in the targets as critical variables when identifying the data-dependent code, while not all variables used in targets are important for triggering the vulnerabilities. For example, to trigger an index overflow vulnerability, only the array and the index are critical.

We did not particularly resolve the false positives in our current implementation because our design has ruled out a large portion of code from exploration. However, advancing the general static analysis techniques could definitely be helpful. We plan to integrate other advanced static analyses (*e.g.*, structure-sensitive points-to analysis in [55]) into SELECTFUZZ to address this limitation.

Solving Complex Path Constraints. SELECTFUZZ uses random mutations to generate inputs and is not effective in solving some complex path constraints. According to our evaluation, it failed to trigger some crashes (*e.g.*, No. 18 in Table I) because of this.

Solving complex path constraints to reach deep code locations is a common challenge in fuzzing. Recent works have proposed techniques like taint tracking [45–47], symbolic executions [48, 56], and structure-aware mutations [57] to assist the fuzzers to generate high-quality inputs. SELECTFUZZ could be integrated with these techniques to improve its ability to solve complex path constraints.

Identifying Vulnerable Code Paths. Similar to other directed fuzzers [2, 16], SELECTFUZZ attempts to explore all the reachable paths to the targets. However, since not all reachable paths to the target code are vulnerable, SELECTFUZZ might extensively explore some reachable yet safe program paths. In our evaluation, it used much time to trigger some crashes (*e.g.*, No. 5 in Table I) because of this reason. One possible orthogonal approach to mitigating this problem is assisting SELECTFUZZ with additional vulnerability information (*e.g.*, the crash dump) besides the target code locations. We leave this as a future work.

[4]https://poppler.freedesktop.org/

VIII. RELATED WORK

In this section, we discuss the closely related works.

Improving Directed Fuzzing Efficiency. Many works have been proposed to improve directed fuzzing efficiency. ParmeSan performed data flow analysis to identify the input bytes affecting the conditionals to targets [27] and AFLChurn used ant colony optimization to evaluate the impacts of input bytes [3]. They mutated some important input bytes to efficiently reach the target locations. The other direction to improve efficiency is identifying the interesting inputs for further mutation. AFLGo firstly introduced the distance metric and assigned more energy to the closer inputs [16]. Hawkeye [1] proposed the augmented adjacent-function distance to avoid the bias to certain traces. Windranger [6] focused on the deviation basic blocks for input distance calculation. Recently, researchers also proposed techniques to filter out unreachable inputs or code paths. Specifically, FuzzGuard trained a deep learning model to discard unreachable inputs [17] and Beacon adopted a lightweight static analysis to prune unreachable paths [2].

SELECTFUZZ adopts selective path exploration by instrumenting only the relevant code. This is different from all prior works that either explore the whole program or the reachable program paths. We believe that our technique is generic and it could be integrated with existing techniques to further improve fuzzing efficiency.

Targeted Program Analysis. Apart from directed fuzzing, some other techniques also adopt the targeted analysis strategy. Directed symbolic execution performs a (heavy-weight) program analysis and constraint solving to generate inputs that reach the target code [33, 58, 59]. Chen *et al.* proposed selective taint analysis that instrumented taint logic only for the instructions that might be tainted [60]. The selective taint approach could improve dynamic taint analysis efficiency by 1.7× in their evaluation [60]. In this work, we applied selective code instrumentation in directed fuzzing and the results showed it achieved significant performance improvement.

IX. CONCLUSION

In this work, we present SELECTFUZZ, a new directed fuzzer that selectively explores program paths for efficient crash reproduction and vulnerability detection. SELECTFUZZ identifies and instruments only the code relevant to reaching or triggering the specified potential vulnerabilities. It also adopts a new distance metric that accurately measures the reaching probability of different program paths and inputs. We evaluated SELECTFUZZ with a set of known vulnerabilities and showed that it achieved up to 46.41× speedup compared to AFLGo. It also performed the best in reproducing eight crashes on the Google Fuzzer Test Suite. We further demonstrate that SELECTFUZZ is complementary to existing techniques like path pruning. Finally, with SELECTFUZZ we detected and reported 14 new vulnerabilities in real-world programs; 11 had been fixed because of our work.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their helpful suggestions and comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14210219).

REFERENCES

[1] H. Chen, B. Chen, Y. Xue, X. Xie, Y. Liu, Y. Li, and X. Wu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[2] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon : Directed grey-box fuzzing with provable path pruning," in *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2022.

[3] X. Zhu and M. Bohme, "Regression greybox fuzzing," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Korea, Nov. 2021.

[4] M. Soltani, A. Panichella, and A. van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1294–1317, 2020.

[5] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, USA, Oct.–Nov. 2017.

[6] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA, May 2022.

[7] Y. Zheng, Z. Song, Y. Sun, K. Cheng, H. Zhu, and L. Sun, "An efficient greybox fuzzing scheme for linux-based iot programs through binary static analysis," in *International Performance Computing and Communications Conference*, 2020.

[8] F. Dong, C. Dong, Y. Zhang, and T. Lin, "Binary-oriented hybrid fuzz testing," in *International Conference on Software Engineering and Service Science*, 2015.

[9] W. Wang, D. Tian, R. Ma, H. Wei, Q. Ying, X. Jia, and L. Zuo, "Shfuzz: A hybrid fuzzing method assisted by static analysis for binary programs," *China Communications*, 2021.

[10] M.-d. Nguyen, R. Bonichon, I. O. Tweag, and R. Groz, "Binary-level directed fuzzing for use-after-free vulnerabilitie," in *International Symposium on Research in Attacks, Intrusions and Defense*, 2020.

[11] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "Symsan : Time and space efficient concolic execution via dynamic data-flow analysis," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2022.

[12] G. Lee and B. Lee, "Constraint-guided directed greybox fuzzingg," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[13] J. Peng, F. Li, B. Liu, L. Xu, B. Liu, K. Chen, and W. Huo, "1dvul: Discovering 1-day vulnerabilities through binary patches," in *Proceedings of the 2019 International Conference on Dependable Systems and Networks (DSN)*, Portland, OR, USA, Jun. 2019.

[14] V. Wüstholz and M. Christakis, "Targeted greybox fuzzing with static lookahead analysis," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, Seoul, Korea, Jun.–Jul. 2020.

[15] L. Zhang, K. Lian, H. Xiao, Z. Zhang, P. Liu, Y. Zhang, M. Yang, and H. Duan, "Exploit the last straw that breaks android systems," in *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2022.

[16] M. Pham, V. Thuan, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, USA, Oct.–Nov. 2017.

[17] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2020.

[18] M. Zalewski, "american fuzzy lop," 2021, https://github.com/google/AFL.

[19] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[20] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time temporal logic guided greybox fuzzing," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA, May 2022.

[21] S. Canakci, N. Matyunin, K. Graffi, A. Joshi, and M. Egele, "Targetfuzz: Using darts to guide directed greybox fuzzers," in *Proceedings of the 17th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, Nagasaki, Japan, Apr. 2022.

[22] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "$razor$: A framework for post-deployment software debloating," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, USA, Aug. 2019.

[23] A. Quach, A. Prakash, and L. Yan, "Debloating software through {Piece-Wise} compilation and loading," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[24] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2019.

[25] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Korea, Nov. 2021.

[26] "Google's fuzzer-test-suite," 2022, https://github.com/google/fuzzer-test-suite.

[27] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan : Sanitizer-guided greybox fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2020.

[28] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2020.

[29] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2018.

[30] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard : Identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montréal, Canada, May 2019.

[31] X. Zhu and M. Bohme, "Regression greybox fuzzing," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Korea, Nov. 2021.

[32] Y. Yu and S. Gan, "Hdbfuzzer – target-oriented hybrid directed binary fuzzer," in *International Conference on Computer Science and Application Engineering*, 2021.

[33] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Korea, Nov. 2021.

[34] J. Ye, R. Li, and B. Zhang, "Rdfuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation," *Mathematical Problems in Engineering*, 2020.

[35] "Aflgo: Directing afl to reach specific target locations." 2013, https://groups.google.com/forum/topic/afl-users/.

[36] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[37] Q. Wu, Y. He, S. McCamant, and K. Lu, "Precisely characterizing security impact in a flood of patches via symbolic rule comparison,"

in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2020.

[38] J. M. P. Cardoso and P. C. Diniz, "Modeling loop unrolling: Approaches and open issues," in *Proceedings of the International Workshop on Embedded Computer Systems*, 2004.

[39] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[40] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2020.

[41] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, K. Bertino, Elisa anf Lu, and J. Li, "Goshawk : Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2022.

[42] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, USA, Aug. 2014.

[43] "Beacon docker image," 2022, https://hub.docker.com/r/yguoaz/beacon/.

[44] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2020.

[45] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2018.

[46] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata : Fuzzing with path aware taint analysis," in *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2022.

[47] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2020.

[48] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[49] "Aflchurn benchmark," 2022, https://github.com/aflchurn/aflchurnbench/tree/main/benchmarks.

[50] "Angora in boringssl," 2022, https://github.com/AngoraFuzzer/Angora/pull/74/files.

[51] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[52] MITRE, "Cve database," 2022, https://cve.mitre.org/.

[53] "Cve-2018-20662," 2018, https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-20662.

[54] Google, "Oss-fuzz," 2021, https://google.github.io/oss-fuzz/.

[55] G. Balatsouras and Y. Smaragdakis, "Structure-sensitive points-to analysis for c and c++," in *Springer Berlin Heidelberg*, 2016.

[56] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller : Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2016.

[57] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "Ems : History-driven mutation for coverage-based fuzzing," in *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2022.

[58] P. D. Marinescu and C. Cadar, "Katch : High-coverage testing of software patches categories and subject descriptors," in *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Saint Petersburg, Russia, Aug. 2013.

[59] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser : a guided fuzzer to find buffer overflow vulnerabilities," in *Proceedings of the 22nd USENIX Security Symposium (Security)*, Washington, DC, USA, Aug. 2013.

[60] S. Chen, Z. Lin, and Y. Zhang, "Selectivetaint: Efficient data flow tracking with static binary rewriting," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.