SDFuzz: Target States Driven Directed Fuzzing

Penghui Li
The Chinese University of Hong Kong
Zhongguancun Laboratory

Wei Meng
The Chinese University of Hong Kong

Chao Zhang
Tsinghua University
Zhongguancun Laboratory

Abstract

Directed fuzzers often unnecessarily explore program code and paths that cannot trigger the target vulnerabilities. We observe that the major application scenarios of directed fuzzing provide detailed vulnerability descriptions, from which highly-valuable program states (*i.e.*, target states) can be derived, *e.g.*, call traces when a vulnerability gets triggered. By driving to expose such target states, directed fuzzers can exclude massive unnecessary exploration.

Inspired by the observation, we present SDFuzz, an effective directed fuzzing tool driven by target states. SDFuzz first automatically extracts target states in vulnerability reports and static analysis results. SDFuzz employs a selective instrumentation technique to reduce the fuzzing scope to the required code for reaching target states. SDFuzz then early terminates the execution of a test case once SDFuzz probes that the remaining execution cannot reach the target states. It further uses a new target state feedback and refines prior imprecise distance metric into a two-dimensional feedback mechanism to proactively drive the exploration towards the target states.

We thoroughly evaluated SDFuzz on known vulnerabilities and compared it to related works. The results show that SDFuzz could improve vulnerability exposure capability with more vulnerability triggered and less time used, outperforming the state-of-the-art solutions. SDFuzz could significantly improve the fuzzing throughput. Our application of SDFuzz to automatically validate the static analysis results successfully discovered four new vulnerabilities in well-tested applications. Three of them have been acknowledged by developers.

1 Introduction

Directed grey-box fuzzing (DGF) usually drives the testing towards highly-valuable target site locations. Though being widely used in crash reproduction and vulnerability validation [3, 7, 16, 26], one primary problem of prior directed fuzzers is that they often explore a large number of program code and paths that cannot trigger the crashes. In the exploration stage of DGF, most existing approaches follow AFLGo's solution [26] and use coverage feedback for expanding the exploration of different code areas. Many code areas are irrelevant and unrequired for reaching the target sites [32]. Such code areas are still unnecessarily explored. In the exploitation stage, prior directed fuzzers utilize the distance metrics based on the control-flow graph (CFG) and call graph (CG) [3,7,26]. The distance metrics do not consider the path conditions, such as control-flow and data-flow conditions. As a result, executions that cannot trigger the vulnerabilities at the target sites because of unsatisfiable control-flow or data-flow conditions might still gain short distances and be overly favored [14, 16, 43]. Excessive testing efforts are thus wasted [43].

A solution to mitigate the above-mentioned problem is to first identify the required program code/paths for triggering the crashes and fuzz only the required one(s). SieveFuzz analyzes the inter-procedural control-flow graph (ICFG) to identify required functions for reaching the target sites and terminates the executions once they reach unrequired functions [32]. Beacon computes the preconditions to reach the target sites via a backward interval analysis and early terminates the executions that fail to satisfy the preconditions [14]. SelectFuzz [20] statically identifies the control- and data-dependent code to the target sites. However, they over-approximate the set of allowed program code and paths, significantly restricting their performance. This is because their analysis is conservative. They only consider the control-flow reachability but fail to analyze other conditions for triggering the vulnerabilities, *e.g.*, the expected reaching order of the target sites [16]. Hawkeye [3] and CAFL [16] include call traces in their designs of distance metrics. However, they do not use them to exclude unrequired code/paths.

We mitigate the problem of unnecessary exploration in DGF with a new finding of *target states, which include the expected call traces and reaching order of the target sites for triggering the vulnerabilities.* In particular, the major tasks of

DGF, such as crash reproduction and vulnerability validation, all provide detailed descriptions of vulnerabilities, such as the target site locations and the associated vulnerability information, *e.g.*, crash dumps, backtraces [10], and source-sink flows [35]. Target site locations describe where the vulnerabilities would occur; the vulnerability information additionally explains the interesting program states (namely target states) about how to trigger the vulnerabilities. The goal of directed fuzzing is not only to reach the target site locations but also to dynamically expose the vulnerabilities. Inspired by this, we aim to drive the fuzzing towards not only the target sites but also the interesting target states. *Thus, besides excluding the exploration that cannot reach target sites, we further avoid the unnecessary executions that cannot reach the target states*.[1]

We thereby design SDFUZZ, a target State driven Directed Fuzzer. We first formalize the target state of a vulnerability as an ordered list of program execution events (*i.e.*, the stack of function invocations at the vulnerability site). SDFUZZ then automatically extracts the target states from vulnerability reports and static analysis results. It takes a selective instrumentation technique to reduce the fuzzing scope to the code required for reaching the target states. In particular, based on the functions in the target states, SDFUZZ analyzes the calling relationship in the inter-procedural control-flow-graph (ICFG) to identify (un)required code. Unlike prior works that directly terminate all executions on unrequired code (identified based on target sites), SDFUZZ still preserves all code of the software but instead removes the code coverage feedback from unrequired code (identified based on target states). The unrequired code is hidden from the fuzzer's perspective, SDFUZZ thus would not assign excessive energy to overly test it. This design choice gets rid of the acute side effects of false code elimination and is fault tolerant: executions can go through (wrongly identified) unrequired code to target sites without being (wrongly) terminated.

SDFUZZ early terminates the execution of a test case once it probes that the remaining execution cannot lead to the target states. One challenge we face is *how to determine if an execution cannot ultimately lead to the target states without completing the execution*. We solve the problem by monitoring runtime program state. At runtime, SDFUZZ maintains the runtime program state, periodically compares it to the target state, and probes if there is any deviation. Situations with any deviation that *cannot be recovered in the subsequent execution of a test case* can be terminated immediately. A deviation can be recovered if there is a program path on the ICFG, through which the deviation function call can possibly be updated to the expected function call specified in the target state.

Besides reducing resource consumption by avoiding unnecessary exploration, SDFUZZ further employs a two-dimensional feedback mechanism to *proactively* drive the testing towards target states. In the first dimension, SDFUZZ calculates a novel target state feedback by computing a similarity score from the best runtime program state of a fuzzing trial to the target states. This dimension aims to favor test cases with better runtime program states. In the second dimension, SDFUZZ refines the widely-adopted distance metrics in DGF. Instead of using an empirically configured constant coefficient (*e.g.*, 10 in AFLGo [26]) as the inter-procedural distance between functions in CG, SDFUZZ uses a new weighted inter-procedural distance by approximating the chance for the caller function to invoke the callee function. The two-dimensional feedback is then used for the power scheduling and seed selection.

We thoroughly evaluated SDFUZZ to assess its capability. We first show that SDFUZZ could generate target states for the majority of bugs in the Magma dataset [13,21], demonstrating its applicability to real-world cases. Besides, on 45 known vulnerabilities, we compared SDFUZZ to four related state-of-the-art directed fuzzers, *i.e.*, AFLGo [26], WindRanger [7], Beacon [14], and SieveFuzz [32]. The results show that SDFUZZ exposed eight, seven, ten, and four more vulnerabilities, respectively, and achieved an average[2] speedup of 2.83×, 2.65×, 1.29×, and 1.81×, respectively, above them. SDFUZZ achieved the best performance for 77.8% of the evaluated vulnerabilities. Our ablation study confirmed the essence of target states to the performance of SDFUZZ. It also showed the benefit of each technique in SDFUZZ, especially the selective instrumentation and execution termination techniques. For instance, on average, SDFUZZ eliminated 48.18% of unrequired functions and early terminated 56.23% of executions. Furthermore, we applied SDFUZZ to automatically extract target states from the results of a static analysis tool—SVF [35]—and validate the results. SDFUZZ successfully identified four new vulnerabilities in three well-tested file processing applications (*e.g.*, `libjpeg`). Three of them have been promptly acknowledged by the developers. We plan to open-source SDFUZZ to facilitate future research.

In summary, this paper makes the following contributions:

- We proposed a new concept of target states and demonstrated their benefits for DGF.
- We designed SDFUZZ, an effective directed fuzzing system driven by target states.
- We incorporated DGF with static analysis to fully automatically validate the results.
- With SDFUZZ, we discovered four new vulnerabilities.

2 Background and Motivation

In this section, we present the background of directed grey-box fuzzing and its representative use examples. We then analyze existing DGF approaches to motivate our solution.

[1] By the term of reaching target states, we mean the execution could exhibit such target program states.

[2] The average values are computed using the geometric mean instead of arithmetic mean throughout this paper.

2.1 Directed Grey-Box Fuzzing

DGF differentiates itself from general coverage-guided grey-box fuzzing in specializing in testing specific code locations. DGF employs an exploration stage driven by the code coverage feedback to expand the covered code locations. Additionally, DGF has an exploitation stage, where past practices mainly employ distance metrics for providing runtime feedback to guide testing direction [7, 16, 26]. For instance, AFLGo defines the distance of a basic block as the harmonic mean of its shortest path lengths to all target sites [26]. In a shortest path, the path length is the sum of the weighted inter-procedural CG distance from the caller function to the target function and the intra-procedural CFG distance from the basic block to the call site's basic block. During a fuzzing trial, a directed fuzzer obtains the execution trace of a test case and computes the average distance of the executed basic blocks as the seed distance. It then uses the distance for seed selection and power scheduling.

2.2 Use Examples of DGF

Directed fuzzers can generally be used to test particular target sites. We show some representative use examples of DGF below.

Reproducing crashes. Software developers often accept vulnerability reports from users. Automated crash report systems [11] collect crash reports from end users and send them to the developers. Since the crash inputs usually convey sensitive user data (*e.g.*, confidential cookies for browser crashes), such systems by default do not attach the crashing proof-of-concept (PoC) inputs [6]. They instead often provide the crash dumps, which contain the crash types and locations, and the involved call traces when the crashes occur. DGF is then helpful for the developers to reproduce the crash specified in the crash dumps. Even if PoC inputs are available, developers may still want to use DGF to comprehend the vulnerabilities and patches.

DGF can address the limitations of symbolic execution based crash reproduction tools. For instance, Star [4], JCharming [28], and BugRedux [15] perform symbolic execution on the slices to target sites. However, they usually collect non-linear, incomplete, and complex path constraints and do not handle external environments, rendering PoC generation through constraint solving difficult [41]. DGFs can address such complex issues by efficient input mutation.

Validating vulnerabilities. Static analysis screens the source code of the program and outputs potential vulnerabilities. The analysis results commonly include vulnerability locations and assumed suspicious flows for the vulnerabilities [29,33,34,39]. To reduce the huge number of false positives, developers can leverage DGF to proactively find PoCs on the suspicious flows and validate the results. Instead of considering all possible flows and paths, a desired DGF ought to focus on only the suspicious flows in the results.

2.3 Existing Approaches and Limitations

Directed fuzzers often unnecessarily test code and paths that cannot trigger the vulnerabilities. This occurs in both the exploration and exploitation stages. We illustrate it with a motivating example shown in Figure 1. There is an assertion failure at `L20`, which is often set as the target site in DGF. The execution through `L4` in function `main()` (namely execution ①) can reach the target site and possibly trigger the assertion failure. The execution going through `L6` (namely execution ②) can only reach the target site.

Unnecessary exploration stage testing on unrequired code. DGF has the goal of reaching target sites and triggering the vulnerabilities there. Its exploration stage directly adopts the conventional coverage metric to expand the covered code, which would favor the executions/test cases that increase the coverage (*i.e.*, new code discovery). However, a large proportion of code is not required for reaching the target sites and would negatively interfere the exploration direction of DGF. For instance, the function `clean()` is not helpful for reaching `L20`.

Unnecessary exploitation stage testing on (un)reachable executions. Even after narrowing down the fuzzing scope to the required code, the exploitation stage still causes unnecessary resource consumption on both executions that can or cannot reach the target sites, *i.e.*, reachable and unreachable executions. As we mentioned in §2.1, DGFs use distance metrics in the exploitation stage [3, 16, 26]. The distance metrics are simply based on the ICFG without analyzing path conditions. Thus paths with unsatisfiable conditions can be regarded as feasible ones. The corresponding executions might obtain short distances and get overly favored.

2.3.1 Prior Mitigations

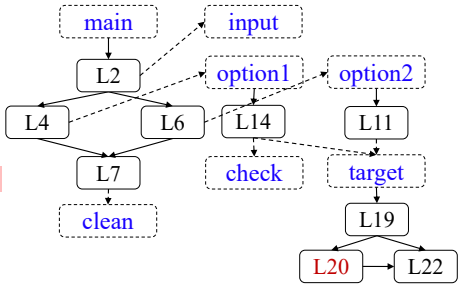
A solution to mitigate this problem is to identify the unrequired code/paths for triggering the vulnerabilities. Beacon [14] statically identifies unrequired code based on the reachability to target site locations. It immediately terminates executions through assertions once the executions trigger unrequired code. SieveFuzz [32] further integrates a dynamic analysis to identify unrequired code and then accordingly terminates executions. SelectFuzz [20] identifies the data- and control-dependent code to the target sites. However, they merely leverage the reachability to the target site locations to drive their required code identification. They could not sufficiently exclude other unrequired code.

Beacon also inserts assertion checks to early terminate executions unsatisfying the preconditions for reaching the target sites through a backward interval analysis. However, the current design of Beacon has two foundational problems. First, to terminate the executions early, the backward interval

```
1  void main() {
2      int x = input();
3      if(x < 10)
4          option1(x); //(1)
5      else
6          option2(x); //(2)
7      clean();
8  }
9
10 void option2(int opcode) {
11     target(opcode);
12 }
```

```
13 void option1(int opcode) {
14     check();
15     target(opcode + 5);
16 }
17
18 void target(int arg) {
19     if (arg <= 20) {
20         assert(arg < 5);
21     }
22     ...
23 }
```

(a) Code example.



(b) ICFG.

Figure 1: A motivating example. The starting line number of a basic block is used as its name in Figure 1(b).

analysis produces complex preconditions, which introduce significant runtime overhead [32]. Second, the preconditions merely embed the control-flow reachability to the target sites without considering additional important information such as call traces and the reaching order of the targets [16]. As a result, Beacon would not terminate reachable executions like execution ②.

Summary. The majority of distance-based DGFs do not leverage any method to exclude unnecessary exploration. They unnecessarily test code and paths in both the exploration and exploitation stages. Prior mitigation approaches [14, 20, 32], however, only consider the reachability to the target sites, yet still keep many unnecessary executions. In this work, we tackle these problems and improve the effectiveness of DGF by exploiting target states.

3 Target States

DGFs specify the target sites—code locations of interest. The target sites often come from two sources: 1) vulnerability reports and 2) static analysis results. We find that from the two sources, we can additionally identify target states, which include the expected call traces and reaching order of the target sites. The target states are helpful for improving directed fuzzing.

3.1 Vulnerability Reports

Software developers often accept vulnerability reports from users. Most vulnerability reports in the wild contain associated crash dumps to help developers confirm the vulnerabilities timely. For example, Figure 2 presents the crash dump (call trace) for the assertion failure in Figure 1. It captures a list of function calls that are currently active in the thread and the invocation locations when the crash is triggered. The `libc` in the crash dump means the library that starts the execution of the main function. Prior DGFs often set the erroneous locations at the top of the crash dump (*i.e.*, crash points) as the

```
1 0x... in target      file.c:20
2 0x... in option1     file.c:15
3 0x... in main        file.c:4
4 0x... in in __libc_start_main
```

Figure 2: Crash dump.

fuzzing target sites, *e.g.*, `file.c:20` in this example.

However, the crash point is only a part of the crash dump. The crash dump pinpoints a highly-valuable call trace about the reported crash, through which the vulnerability can be easily reproduced. With only the target site locations, directed fuzzers would unnecessarily explore many infeasible program states (*e.g.*, execution ②). Driving a directed fuzzer to invoke the list of functions conforming to the crash dump could enable it to trigger the vulnerability quickly. Furthermore, for multi-target vulnerabilities (*e.g.*, use-after-free), there can be associated call traces for each involved target site, and their expected reaching order can be inferred [16]. *In this work, we define the expected call traces and the reaching order of target sites as target states.* No prior directed fuzzer has used the target states to eliminate unnecessary exploration. Prior solutions [14, 32] only use the target site locations and partially exclude exploration that could not reach the target sites.

Crash dumps are available in most of the vulnerability reports, from which target states can be extracted. We studied four popular file processing applications such as `swftophp` and `lrzip`, to which fuzzing has been widely applied [14, 26]. We examined all historical crashes in the applications and summarized an unbiased dataset of 259 crashes. We found that crash dumps existed in 191 (73.75%) cases. This demonstrates that target states can be obtained in most cases. A solution that requires the target states still has wide applicability.

3.2 Static Analysis Results

Static analysis is based on heuristics and often outputs detailed information for analysts to comprehend the analysis

results. Static analysis tools report the vulnerable flows to the target sites where the conditions of the heuristics are met. For instance, Joern [39]—a taint analysis tool—would report the vulnerable flow from the untrusted external data sources to the target sites. SVF [34, 35] for memory-safety issues would report vulnerable program flows of improper memory usage.

Prior directed fuzzers often set the memory operations in the reported flows as the target sites. This would make DGF explore all possible paths to the target sites. However, there are abundant flows that do not meet the conditions of the heuristics. Testing them causes non-negligible resource waste. To efficiently validate the vulnerability, a fuzzer ought to focus on exhibiting the vulnerable flows instead of all possible ones. Similar to vulnerability reports, we can derive the expected target states from such commonly provided vulnerable flows in static analysis results.

3.3 Formalization

We define a general representation of program states used in this work, as shown in Formula 1. We learn from examples (*e.g.*, Figure 2) that the crash dumps contain a sequence of ordered function calls and the invocation locations. We thus formalize a program state (PS) as the stack of function invocations (call sites). Each item in the program state is a tuple of the function name and the invocation location (`(Func, Loc)`). The target state (`TS`) of a vulnerability or a bug is the program state when it is triggered. To reach a target state, the functions in it ought to be reached or called in the right order. Therefore, Formula 1 captures the expected call traces and their reaching order. For example, we can formalize the crash dump in Figure 2 as the target state (`TS1`) shown in Figure 4(a). For multi-target vulnerabilities, we can usually derive one target state per target site and sort them to obey the required target reaching order (`TSs`) [16]. Similarly, the target states of static analysis results can be formalized using program states.

$$\begin{aligned} PS &= [(Func_1, Loc_1), (Func_2, Loc_2), ..., (Func_n, Loc_n)] \\ TSs &= [TS_1, TS_2, ..., TS_m] \end{aligned} \tag{1}$$

4 SDFuzz

4.1 Overview

We advance DGF using target states. In the vast exploration space of a program, a significant proportion cannot trigger the target vulnerabilities. Testing all of it would cause unnecessary resource consumption. Fortunately, target states describe interesting program states where the vulnerabilities would (likely) occur. Our intuition is to drive the fuzzing to exhibit these interesting target states instead of merely reaching the target site locations. We use the target states from vulnerability reports and static analysis results to *exclude unnecessary exploration that cannot reach the target states*. We further design new techniques to proactively guide the testing. Hawkeye [3] and CAFL [16] also use call traces in their distance design to guide the exploration. They still consider the whole exploration space of a program. SDFuzz starts from a different angle by proactively removing unnecessary testing.

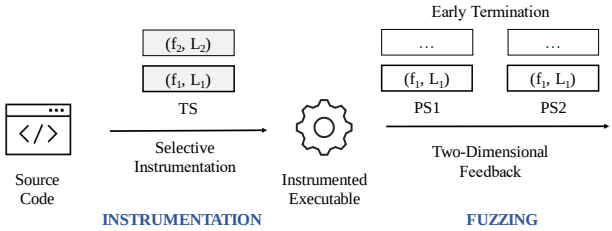


Figure 3: The workflow of SDFuzz.

We thereby develop a new directed fuzzing system, SDFuzz, based on the target states. The workflow of SDFuzz is depicted in Figure 3. SDFuzz first automatically extracts target states and parses them into specified formats. SDFuzz then identifies the required code for reaching the target states and removes other unrequired code from fuzzing. It particularly advances prior solutions [14, 32] in using the target states instead of target sites. SDFuzz early terminates the executions once it probes that the remaining execution of a test case cannot reach the target states, thereby increasing the fuzzing throughput (*i.e.*, number of executions per unit time). SDFuzz uses a two-dimensional feedback mechanism to proactively guide the testing towards the target states. In the first dimension, SDFuzz measures the similarity between the best runtime state of a test case and the target states, and favors the ones with higher similarity. In the second, SDFuzz adopts a new precisely-weighted inter-procedural distance metric.

4.2 Extracting Target States

To extract the target states, SDFuzz requires either vulnerability reports or static analysis results. For the latter, SDFuzz employs an existing static analysis tool to analyze the program's source code.

Vulnerability reports. The crash dump consists of the records of the active function calls when a vulnerability is triggered, as shown in Figure 2. Each record contains: 1) function name (*e.g.*, `option1`) and 2) invocation location (*e.g.*, `file.c:15`). Therefore, we first search the vulnerability reports about descriptions containing such information using regular expressions. After extraction, we further parse them to decide if they match the formats defined in Formula 1. We also automatically sort the target states based on the vulnerability types and the descriptions. For example, a use-after-free vulnerability often contains multiple target states. We would

Algorithm 1: Required code identification.

```
   input  : TSs, ICFG
   output : requiredFuncs
1  initRequiredFuncs ← [ ]
2  requiredFuncs ← [ ]
3  for TS ∈ TSs do
4      for f ∈ TS do
5          initRequiredFuncs.insert(f)
6          funcs ← backwardAnalysis(f, ICFG) // get
             functions with intra-procedural dependencies
7          initRequiredFuncs.insert(funcs)
8      end
9  end
10 while ! initRequiredFuncs.empty() do
11     f ← initRequiredFuncs.remove()
12     if f ∉ requiredFuncs then
13         requiredFuncs.insert(f)
14         callees ← getCallees(f, ICFG) // get callees
             of f
15         initRequiredFuncs.insert(callees)
16     end
17 end
18 return requiredFuncs
```

sort the target states sequentially by the free and the use site.

Static analysis results. SDFuzz also automatically extracts the target states from the static analysis results. Since different static analysis tools take diverse ways to represent their results, naturally, the automated extraction has to be specially designed for each static analysis tool. We currently develop SDFuzz to coordinate with one popular static analysis tool, SVF [34, 35]. To the best of our knowledge, target states can be extracted from other static analysis tools such as CodeQL [9] and Joern [39] with only additional efforts to parse the results. The program's source code is usually required for running such static analysis tools.

4.3 Selective Instrumentation of Required Code

SDFuzz reduces the fuzzing scope by selectively instrumenting only the required code for coverage feedback in the exploration stage of DGF. SDFuzz first identifies what part of the code is required and then deliberately excludes the other unrequired code from the fuzzing process. Our solution preserves the code required for reaching the target states, which is a subset of the code for reaching the target sites as preserved in SieveFuzz and Beacon. This is because the target states further constrain the paths to reach target sites. It thus can help filter out much more code and improve fuzzing throughput. Our solution selectively instruments the required code for code coverage feedback instead of directly removing it from the source or executable.

We propose a function-level algorithm shown in Algorithm 1 to identify the required code. It takes as inputs a set of target states (`TSs`) and the ICFG of the target program (`ICFG`). The functions appearing in the target states (namely target state functions) are related to the vulnerabilities and are directly included as the required functions (line 5). Besides, these target state functions might depend on other functions. Our algorithm first performs a backward intra-procedural analysis to identify the functions that a target state function depends on (line 6). A function is included if there is an intra-procedural path between the basic block that has a function call site and the basic block of a target state function. For instance, function `check()` is included because function `target()` at `L15` depends on it. Additionally, those newly included functions might invoke other functions to accomplish their functionalities. Therefore, our algorithm analyzes the CG and includes those functions on the CG paths outing from the initially included functions (lines 14-15). In this way, SDFuzz expands the set of functions required for realizing the target states. The callee functions of `check()` are added to the required code for this.

Instead of directly removing the code from the target executable, SDFuzz employs an instrumentation-based method to exclude unrequired code. We find that DGF requires collecting code coverage feedback, which reveals the existence of code areas. SDFuzz thus selectively instruments only the identified required code for code coverage feedback, which hides the other unrequired functions from the fuzzer and reduces the fuzzing scope. This design is fault-resilient. Even if some code areas are wrongly identified as unrequired, executions can still go through such code areas to further approach the target sites and states. SDFuzz would not assign testing energy to explore the paths that are not instrumented. Thus it gets rid of the critical downside caused by false code elimination in prior solutions [14, 32]. It also reduces the overhead caused by the instrumented coverage tracking code.

4.4 Early Termination of Executions

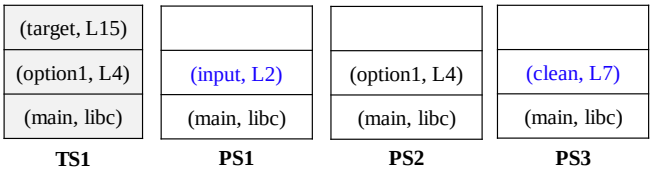
We develop a new fuzzing technique that early aborts the executions that cannot reach the target states. If some executions are known to be unable to reach the target states, we terminate them early to save the exploration resources. This can dramatically increase the fuzzing throughput by terminating unnecessary executions early. Unlike prior reachability-based execution termination approaches [14, 32], SDFuzz also terminates reachable executions that cannot reach the target states.

To perform the early termination of executions, we have to predict if the execution can ultimately reach the target states or not. This is difficult because the program state is dynamically updated along the program execution, *e.g.*, via function invocations and returns. Given the high complexity of modern programs, the program state space they can exhibit can be huge [1].

```
TS1=[(main, libc), (option1, L4), (target, L15)]

PS1=[(main, libc), (input, L2)]
PS2=[(main, libc), (option1, L4)]
PS3=[(main, libc), (clean, L7)]
```

(a) A target state and selected program states.



(b) Target state and selected program states, with the root deviations are in blue.

Figure 4: A target state and selected program states.

Runtime program state monitoring. SDFuzz monitors the runtime function invocations and records the stack of function invocations. These functions are pushed or popped from the stack with function invocations or returns. The function invocation locations enable SDFuzz to distinguish the same function invoked at different locations. Program state tracking might potentially lead to state explosion [38] and can cause heavy overhead. We mitigate this issue by tracking the states for only the functions relevant to the target states. In particular, SDFuzz only updates and checks the program state for early termination when the program calls or returns from the functions in target states.

An unrecoverable deviation based solution. As mentioned in §3.3, the target states for a multi-target vulnerability are an array of ordered lists of function call invocations, each for a target site. Our algorithm (shown in Algorithm 2) thus takes as inputs the current program state (`PS`) at a point in time, previously reached target states (`reachedTSs`), the ordered target states (`TSs`), and the `ICFG`. It iterates over the target states to find the first target state that has not been reached during a fuzzing trial of a test case (lines 3-6). If all target states have been reached, the algorithm directly returns (lines 7-8). Otherwise, it then checks the deviation function calls, especially the first deviation—*root deviation* through `rootDeviation` function (line 10). The root deviation denotes where the program state starts to deviate from the unreached target state. This is done by iteratively comparing the call sites (lines 20-26) to find the first deviation.

If there is any deviation (line 11), our algorithm further measures if the remaining execution can recover the deviations to reach the target state based on the ICFG (line 12). If an execution has unrecoverable deviations in its program state, it can be immediately terminated. Our algorithm checks the ICFG of the program and probes if there is a program path from the root deviation code location to the expected function call in the target state. Such a path means the deviations might be recovered in the future execution because the execution can return from the root deviation function call and run to the expected one. Accordingly, the execution that might recover the deviation would not be terminated.

We illustrate the workflow of our algorithm with three program states of an execution (namely `PS1`-`PS3`) listed in Figure 4. `PS1` is observed when the execution reaches right after line 2. The program state deviates from the target state (*i.e.*, `TS1`) in the second item, *i.e.*, `(input, L2)` *v.s.* `(option1, L4)`. The deviation is possibly recoverable because subsequent execution might return from the function `input()` and run next to the expected function `option1()` at `L4`. From the perspective of ICFG, this can be reflected as the existence of a program path from the deviation location (*e.g.*, `L2`) to the expected one (*e.g.*, `L4`). Thus the execution would not be terminated at `PS1`. At `PS2`, the program state is exactly the prefix of `TS1` without additional deviations and does not deviate from `TS1`. The execution would not get terminated. However, in the case of `PS3`, it deviates at `(clean, L7)` against the `(option1, L4)` in `TS1`, and there is no path from `L7` to `L4`. The execution would get terminated. Beacon's solution [14], on the other hand, is not able to terminate the execution in the middle.

4.5 Two-Dimensional Feedback

4.5.1 Target State Feedback

SDFuzz also compares the runtime program states to the target states and computes a similarity score to proactively guide the exploration. The feedback favors test cases with more similar program states to the target states. Some recent works such as CAFL [16], LOLLY [17], and Hawkeye [3] also use the runtime program behaviors. However, the mechanism in SDFuzz is tailored to target states and considers calling contexts.

The workflow is also shown in Algorithm 2. After finding the first unreached target state (`nextTS`) at lines 3-6, SDFuzz uses the index of the root deviation to compute the similarity score. If the current program state does not fully match the first unreached target state (`nextTS`), SDFuzz first measures how well the current program state fits it by computing the ratio of matched `deviationIdx` over its size (line 13). Our algorithm also considers previously reached target states and sums a score of the ratio and the size of `reachedTSs`. The score is further normalized using the number of target states and returned. If the current program state matches `nextTS`, our algorithm directly returns the proportion of reached target states (line 16). Since the algorithm might be invoked multiple times for the execution of a test case, we assign the best score as the result of the test case.

Algorithm 2: Execution termination and target state similarity.

```
   input  : PS, TSs, reachedTSs, ICFG
   output : score, termination, reachedTSs
1  nextTS ← null
2  termination ← false
3  for TS ∈ TSs\reachedTSs do
4  |  nextTS ← TS // find next target state
5  |  break
6  end
7  if nextTS = null then
8  |  return 1, false, reachedTSs
9  end
10 deviationIdx ← rootDeviation(PS, nextTS)
11 if deviationIdx ≠ nextTS.size then
12 |  termination ← ! ICFG.path.exists(PS[deviationIdx],
   |    nextTS[deviationIdx]) // check if recoverable
13 |  score ← (deviationIdx / nextTS.size + reachedTSs.size)
   |    / TSs.size
14 else
15 |  reachedTSs.insert(nextTS) // no deviation
16 |  score ← reachedTSs.size / TSs.size
17 end
18 return score, termination, reachedTSs
19
20 function rootDeviation(PS, TS):
21 |  index ← 0
22 |  for index<min(PS.size, TS.size) & PS[index]=TS[index]
   |    do
23 |  |  index ← index + 1
24 |  end
25 |  return index
26 end
```

4.5.2 Distance Feedback

SDFuzz also uses a distance metric to guide the fuzzing process. Prior distance metrics are imprecise because they consider every edge in CG equally. They empirically configure a constant weight (*e.g.*, 10 in AFLGo-based directed fuzzers) to approximate the chance for reaching the target functions [3, 26]. Therefore, executions exhibiting long call chains, even with high chances to reach the target functions, are possibly assigned with large distance values and get deprioritized. Distinct functions ought to be evaluated differently.

SDFuzz mitigates the imprecision with precise edge weights when computing inter-procedural distances. The edge weight is expected to reflect the chance for the caller function to invoke the callee function. SDFuzz computes the edge weight based on the call-site weights. We define the *call-site weight* for a caller function to invoke a callee function as the intra-procedural distance from the start of the caller function to the call site of the callee (*i.e.*, the basic block distance on the shortest path as in AFLGo [26]). Since there might be multiple call sites to the same callee function, the inter-procedural *edge weight* is the shortest call-site weight ($weight(f_i, f_j)$) between the caller function $f_i()$ and the callee function $f_j()$. This is also shown in Formula 2, where $d_{f_i}()$ computes the intra-procedural distance in function f_i. For the function `option1()` in Figure 1, since the function start and the call site of the function `check()` reside in the same basic block, their edge weight is `0` instead of `10` as in AFLGo.

$$weight\left(f_i, f_j\right) = min\left(d_{f_i}\left(BB_{f_i-start},\ BB_{f_j-call-site}\right)\right) \tag{2}$$

The edge weights between callers and callees form a weighted CG. This allows SDFuzz to compute precise CG distance between two arbitrary functions. We formalize the method to compute inter-procedural distance in Formula 3. If there is at least one path from function f_s to function f_e in the CG, their distance is calculated as the sum of edge weights in the shortest path. Otherwise, if there is no path from function f_s to function f_e, the distance is considered as not available or infinite. We explain how we construct CG in §5.

$$interDistance\left(f_s, f_e\right) = min\left(\sum_{(f_i,f_j)\in path} weight\left(f_i, f_j\right)\right) \tag{3}$$

4.5.3 Seed Selection and Power Scheduling

SDFuzz incorporates the two dimensions of feedback to guide the seed selection and power scheduling. To drive the fuzzing towards the target states quickly, SDFuzz sorts the seeds in the corpus sequentially by two attributes—target state feedback and seed distance. Generally, SDFuzz prefers seeds with better target state feedback and shorter distances. It uses target state feedback as the primary sorting attribute and distance as the secondary. The reason is that the target state feedback capturing the runtime context is more precise, and could better help approach target states. SDFuzz also refines the power scheduling algorithm of AFLGo to assign energy to the seeds according to the two-dimensional feedback.

5 Implementation

We implement a prototype of SDFuzz to fuzz C/C++ programs. We first use static analysis to build a program representation to facilitate the required code identification and execution termination. We then employ a compile-time analysis to instrument the target program, which inserts the necessary code for tracking coverage, maintaining program states, computing seed distances, *etc.* Besides, we develop a runtime library to retrieve runtime program states and perform selective execution termination. The main fuzzing component was implemented atop AFLGo [26].

Static analysis. We statically analyze the target program to identify the required functions (§4.3). The analysis leverages Andersen's points-to analysis to identify the call targets for indirect calls [31]. Our implementation currently reuses the associated pipeline of SVF [34]. This results in the CFG and CG for our analysis. We also discuss the potential issues of static analysis in §7.

Instrumentation. We maintain the list of interesting functions obtained from §4.3 and selectively instrument them for code coverage feedback. Other functions not in the list are still preserved but are not instrumented for code coverage feedback. As a result, they can still be executed. The instrumentation component is realized by modifying AFL's LLVM compiler.

Runtime state tracking. We implement a runtime library for program state maintenance and selective execution termination. In the library, we track only a set of functions that are relevant to the target states and maintain the stack of function invocations. At runtime, the library periodically compares the program state to the target state(s) to determine early execution termination. Besides, it also scores the runtime program states to provide target state feedback. We modify the `update_bitmap_score()` and `cull_queue()` functions in AFL to achieve our seed selection strategies.

6 Evaluation

We extensively evaluate SDFuzz to answer the following questions.

- **RQ1.** How well can SDFuzz generate target states in practice?
- **RQ2.** What is the capability of SDFuzz in exposing vulnerabilities?
- **RQ3.** How effective can SDFuzz reduce unnecessary exploration?
- **RQ4.** How do the techniques in SDFuzz contribute to its performance?
- **RQ5.** How effective is SDFuzz in discovering new vulnerabilities?

6.1 Target State Generation Capability

We first assess if SDFuzz can automatically extract target states for real-world vulnerabilities. We choose Magma [13, 21], a widely-used fuzzing benchmark containing 138 bugs along with their corresponding reports. We check the crash dumps for the included bugs and then apply SDFuzz to extract target states. After that, we manually verify the correctness of the extracted target states.

The results show that SDFuzz could successfully extract correct target states for 127 out of 138 cases, where the crash dumps are included in the bugs' reports. This suggests the high applicability of SDFuzz for real-world bugs. SDFuzz could not generate target states for the cases without available crash dumps. The vulnerabilities have diverse target states, *e.g.*, the number of targets ranges from one to three, and the number of function invocations ranges from two to six. Empirically, we have not observed an impact of target states on the performance of SDFuzz.

6.2 Performance of SDFuzz

We then evaluate the performance of SDFuzz on a set of known vulnerabilities.

Experimental Setup. We construct a comprehensive dataset. In particular, we include the programs and vulnerabilities evaluated by other recent DGFs [7, 14, 20, 32]. Google Fuzzer Test Suite [12] and AFLGo's Test Suite [27] are also included. In total, we include 45 unique vulnerabilities in our dataset, and we list them in Table 1. Other vulnerabilities evaluated by recent DGFs are excluded from our evaluation mainly because of compilation issues such as obsolete and missing dependencies and incompatible compilation environments. The included vulnerabilities span a comprehensive set of vulnerability types such as buffer overflow, heap overflow, *etc.*, and can well evaluate the capability of SDFuzz. All the experiments are conducted on a server running Ubuntu 18.04 with two 18-core Intel Xeon Gold 6140 CPUs and 256GB RAM.

We prepare the target states and the seed inputs for the experiments. We first find the origin of the vulnerability report and extract the target states. SDFuzz successfully extracted the target states for all the cases. We then use SDFuzz to test the vulnerabilities for five runs, each with a 24-hour time limit. For the vulnerabilities in Google's Fuzzer Test Suite, we use the seed inputs (if available) provided in the repository; we use empty seed inputs for other cases.

Required code identification. Our selective instrumentation technique can significantly reduce the fuzzing scope to the required code. We first analyze the proportion of the required code that SDFuzz identified for the 45 evaluated vulnerabilities. In particular, SDFuzz eliminated 48.18% of unrequired functions on average and narrowed down the fuzzing scope to the other 51.82% of required functions. For several cases (*e.g.*, #24 in re2), SDFuzz could even eliminate over 80% of unrequired functions and trigger the vulnerabilities.

Vulnerability exposure. We measured the time used for exposing the known vulnerabilities and presented the evaluation results in Table 1. SDFuzz could reproduce 44 out of the 45 vulnerabilities within the time limit of 24 hours (1,440 minutes). This demonstrates the high effectiveness of SDFuzz in exposing known vulnerabilities.

6.3 Comparison with Other Approaches

We compared SDFuzz to existing directed fuzzers on the same dataset. We thoroughly investigated the literature and included the state-of-the-art open-sourced directed fuzzers as the comparison targets: AFLGo [26], WindRanger [7], and SieveFuzz [32]. Beacon [14] is publicly available in the form of binary [40]. We additionally used the provided binary and included it in the comparison. We could not add some other related works (*e.g.*, Hawkeye [3], CAFL [16]) mainly because they are not open-sourced. We ran these fuzzers also for five

Table 1: Vulnerability exposure results. `Factor` is the ratio of time used by a tool compared to that of SDFuzz. `CE` denotes compilation error. `TO` denotes that a tool reaches the time limit (timeout) before triggering a vulnerability. The best result of a case is underlined.

ID	Program	Location	AFLGo			WindRanger			Beacon			SieveFuzz			SDFuzz
			Time	Factor	p-val	Time	Factor	p-val	Time	Factor	p-val	Time	Factor	p-val	Time
1	libming	decompile.c:349	216	2.45	0.003	195	2.22	0.002	147	1.67	0.001	199	2.26	0.007	88
2	libming	decompile.c:398	268	1.71	0.008	348	2.22	0.003	194	1.24	0.050	282	1.80	0.030	157
3	LMS	service.c:227	5	1.67	0.009	8	2.67	0.006	3	1.00	0.001	3	1.00	0.001	3
4	mjs	mjs.c:13732	272	1.36	0.132	204	1.02	0.012	128	0.64	0.003	228	1.14	0.023	200
5	mjs	mjs.c:4908	8	2.67	0.007	5	1.67	0.004	5	1.67	0.006	3	1.00	0.001	3
6	tcpdump	print-ppp.c:729	608	4.68	0.004	708	5.45	0.003	CE	-	-	512	3.94	0.003	130
7	lrzip	stream.c:1747	372	18.60	0.005	251	12.55	0.003	38	1.90	0.001	176	8.80	0.003	20
8	lrzip	stream.c:1756	329	7.48	0.002	224	5.09	0.001	158	3.59	0.003	137	3.11	0.009	44
9	objdump	objdump.c:10875	785	5.38	0.002	752	5.15	0.008	235	1.61	0.003	327	2.24	0.003	146
10	objdump	dwarf2.c:3176	TO	-	-	618	7.92	0.001	CE	-	-	154	1.97	0.019	78
11	libssh	messages.c:1001	TO	-	-	TO	-	-	TO	-	-	TO	-	-	1,112
12	libxml2	valid.c:952	151	2.44	0.009	42	0.68	0.004	52	0.84	0.003	70	1.13	0.001	62
13	libxml2	messages.c:1001	217	1.43	0.003	209	1.38	0.002	78	0.51	0.003	192	1.26	0.018	152
14	libxml2	parser.c:10666	134	3.35	0.012	211	5.28	0.007	TO	-	-	78	1.95	0.009	40
15	libarchive	format_warc.c:537	TO	-	-	TO	-	-	TO	-	-	TO	-	-	1,039
16	Little-CMS	cmsintrp.c:642	382	2.98	0.003	565	4.41	0.003	229	1.79	0.001	258	2.02	0.004	128
17	boringssl	asn1_lib.c:459	511	4.26	0.006	368	3.07	0.004	263	2.19	0.003	346	2.88	0.006	120
18	c-ares	ares_create_query.c:196	3	3.00	0.019	3	3.00	0.122	1	1.00	0.151	1	1.00	0.132	1
19	guetzli	output_image.cc:398	42	10.50	0.030	51	12.75	0.003	17	4.25	0.004	25	6.25	0.012	4
20	harfbuzz	hb-buffer.cc:419	TO	-	-	TO	-	-	TO	-	-	1,350	2.13	0.001	633
21	json	fuzzer-parse_json.cpp:50	8	4.00	0.013	19	9.50	0.003	3	1.50	0.001	5	2.50	0.003	2
22	woff	buffer.h:86	519	1.46	0.019	638	1.79	0.003	389	1.09	0.001	443	1.24	0.003	356
23	vorbis	codebook.c:479	TO	-	-	TO	-	-	198	0.78	0.001	TO	-	-	254
24	re2	nfa.cc:532	1,121	12.18	0.005	654	7.11	0.003	157	1.71	0.001	465	5.05	0.005	92
25	pcre	pcre2_match.c:5968	55	4.23	0.001	30	2.31	0.001	8	0.62	0.005	27	2.08	0.005	13
26	tcpdump	in_cksum.c:108	369	1.16	0.104	420	1.32	0.040	CE	-	-	CE	-	-	319
27	tcpdump	print-isakmp.c:2502	615	1.21	0.009	502	0.98	0.053	TO	-	-	419	0.82	0.008	510
28	tiffcp	tiffcp.c:1596	551	3.01	0.012	580	3.17	0.064	319	1.74	0.071	611	3.34	0.050	183
29	tiffcp	tiffcp.c:1423	TO	-	-	TO	-	-	TO	-	-	1,284	1.29	0.091	994
30	imginfo	jpc_cs.c:316	93	2.21	0.022	172	4.10	0.029	25	0.60	0.032	39	0.93	0.012	42
31	imginfo	bmp_dec.c:474	182	1.52	0.010	TO	-	-	116	0.97	0.012	209	1.74	0.019	120
32	imginfo	jas_image.c:378	382	5.23	0.068	273	3.74	0.044	CE	-	-	193	2.64	0.091	73
33	lame	gain_analysis.c:224	91	2.60	0.023	30	0.86	0.033	39	1.11	0.029	58	1.66	0.0.43	35
34	lame	mpglib_interface.c:142	911	2.34	0.023	1,029	2.64	0.012	592	1.52	0.084	671	1.72	0.004	390
35	lame	get_audio.c:1452	488	1.74	0.043	391	1.39	0.045	276	0.98	0.019	401	1.91	0.029	281
36	mujs	jsrun.c:1024	347	1.96	0.003	287	1.62	0.012	591	3.34	0.004	638	3.60	0.004	177
37	mujs	jsdump.c:892	694	2.62	0.009	392	1.48	0.018	TO	-	-	192	0.72	0.008	265
38	mujs	jsdump.c:867	605	3.83	0.018	482	3.06	0.014	291	1.84	0.007	263	1.66	0.007	158
39	mujs	jsvalue.c:396	TO	-	-	TO	-	-	1,109	-	-	1,284	-	-	TO
40	libming	parser.c:3232	118	1.59	0.059	231	3.12	0.091	45	0.61	0.038	83	1.12	0.036	74
41	libming	outputtxt.c:143	372	2.13	0.007	413	2.36	0.009	283	1.62	0.023	309	1.77	0.016	175
42	libming	parser.c:3089	439	2.02	0.034	364	1.68	0.019	492	2.27	0.043	284	1.31	0.024	217
43	libtiff	tif_dirwrite.c:1901	TO	-	-	893	1.52	0.041	482	0.82	0.018	693	1.18	0.075	588
44	libtiff	tif_read.c:346	TO	-	-	TO	-	-	1,034	1.39	0.012	920	1.24	0.056	744
45	libtiff	tiffcp.c:1386	643	1.45	0.043	325	0.74	0.027	339	0.77	0.019	458	1.03	0.032	443

24-hour runs and reported the average vulnerability exposure time. Since these works do not use target states, we acknowledge that our comparison with them can hardly be perfectly fair. Nevertheless, we tried our best to conduct a fair comparison by strictly following the instructions provided by the compared tools, and using the same hardware environments and initial seeds.

Vulnerability detection. The comparison results are shown in Table 1. SDFuzz *generally outperformed other directed fuzzers with more vulnerabilities exposed*. Specifically, AFLGo, WindRanger, Beacon, and SieveFuzz exposed 36, 37, 34, and 40 vulnerabilities, respectively. The numbers of the exposed vulnerabilities are fewer than SDFuzz's. Note that we failed to compile several cases in our evaluation, shown as `CE` in Table 1.

We characterize the time each tool used for triggering the vulnerabilities. SDFuzz *used a shorter time than the compared directed fuzzers in most of the exposed cases*. We compute a factor value as the ratio of the time used by a tool to that of SDFuzz in each case. The factor value describes the performance speedup between tools. The factor value is not available (shown as - in Table 1) if a fuzzer does not trigger the vulnerability. We further compute the average speedup as the geometric mean of the factor values for those exposed vulnerabilities (excluding `CE` and `TO` cases).[3] In general, SDFuzz achieved an average speedup of

[3]Excluding `TO` cases actually under-approximates the speedup factors

2.83×, 2.65×, 1.29×, and 1.81× above AFLGo, WindRanger, Beacon, and SieveFuzz, respectively. SDFuzz also outperformed AFLGo, WindRanger, Beacon, and SieveFuzz by up to 18.60×, 12.55×, 4.25×, and 8.80×, respectively. The best result per vulnerability is underlined in Table 1. We observe that SDFuzz performed the best in 35 out of the 45 cases (77.8%), demonstrating the effectiveness of our new techniques.

We further employed a Mann-Whitney U test [22] on vulnerability exposure time to measure the statistical significance of our experiment results. We find that our results are significant in the majority of the cases under the significance level of 0.05 (*i.e.*, most cases in Table 1 have a p-value less than 0.05). Therefore, given the high diversity of the vulnerabilities in our dataset, we confidently conclude that SDFuzz could trigger vulnerabilities more quickly.

We observe that several vulnerabilities were significantly hard for prior directed fuzzers to trigger, which SDFuzz successfully triggered. For instance, cases #11 and #15 were not exposed by all other four evaluated directed fuzzers; cases #20 and #23 were not triggered by three of the other evaluated directed fuzzers. We investigated the source code and the historical input queue and identified the reason why SDFuzz could more easily trigger them. These cases generally have a significantly large number of paths on the CG from the entry function to the target function(s). A large proportion of them is not feasible dynamically due to the unsatisfiable path constraints. For example, in case #11, a buffer overflow vulnerability, other directed fuzzers could not reproduce it within the time limit. They favored wrong paths and tested this vulnerability in a wrong direction, where the vulnerability-triggering conditions were hardly achieved. SDFuzz, however, chose a feasible path that was more likely to approach the target state. The analysis suggests that the target states provided additional guidance, which directed SDFuzz to trigger the vulnerabilities, whereas the other fuzzers easily got stuck in wrong directions.

Code elimination. SieveFuzz [32] uses a code elimination technique based on target site locations and is open-sourced. We also investigated how well SieveFuzz eliminated code. It removed around 31.53% of unrequired code on average, which is 43.29% less than what SDFuzz eliminated. This demonstrates the benefits of the target state information for code elimination. We were not able to check such statistics for Beacon [14] since the authors only released its binary, and we could not enhance it to obtain such internal statistics.

Path pruning and fuzzing throughput. The effectiveness of path pruning can be reflected in the fuzzing throughput, *i.e.*, number of executions per unit time. We find that SDFuzz *achieved a much higher fuzzing throughput.* Since different programs normally have distinct processing time, we thereby calculate a throughput factor value as the ratio of the throughput of a tool to that of AFLGo in each case. We then compute the geometric mean on all cases as the average throughput. On average, SDFuzz, WindRanger, Beacon, and SieveFuzz had the throughput factor value of 9.32, 0.93, 1.43, and 8.09, respectively. This shows that fuzzers employing execution termination techniques have higher throughput compared to the ones with only distance metrics. For instance, SDFuzz, Beacon, and SieveFuzz had higher throughput than the other two. Besides, SDFuzz, driven by target states, achieved the highest fuzzing throughput in the evaluated vulnerabilities. This could be explained as its execution termination to also abort some reachable executions. As for WindRanger, it underperformed AFLGo mainly because of its taint analysis overhead.

Vulnerability-triggering paths. *We found that other fuzzers mostly triggered the vulnerabilities through the identical paths that* SDFuzz *derived from the target states*. Specifically, we replayed the crashing input that a fuzzer generated to expose a vulnerability and analyzed the triggered program path. We then correlated such paths to the ones in the target states. SDFuzz triggered the vulnerabilities through these paths. On the other hand, AFLGo, WindRanger, Beacon, and SieveFuzz ultimately took the paths in the target states for 28, 20, 25, and 30 cases, respectively. Such an observation has two implications. First, SDFuzz could directly drive the exploration towards such paths without sparing too much effort on other paths. This is the root reason why SDFuzz could have superior performance compared to other directed fuzzers. Second, by driving towards the target states, though SDFuzz could potentially overlook some other paths, this would not significantly undermine the performance of SDFuzz. Therefore, we believe SDFuzz could significantly benefit state-of-the-art crash reproduction.

6.4 Component-Wise Analysis

We conduct an ablation study on the same dataset to understand how each technique in SDFuzz contributes to the performance. First, to assess the effect of the target states, we design a variant of SDFuzz, namely SDFuzz$_{bl}$, that utilizes only the bug locations (target sites)—the reduced target states. We add a variant of SDFuzz—SDFuzz$_{-si}$—by disabling its selective instrumentation. Additionally, we design four variants for the component-wise evaluation. Since SDFuzz is built atop AFLGo, each variant enables one key technique over AFLGo. In particular, AFLGo$_{+si}$, AFLGo$_{+et}$, AFLGo$_{+sf}$, and AFLGo$_{+df}$ further enables selective instrumentation, execution termination, target state feedback, and distance feedback, respectively, atop AFLGo.

We use the same experimental setup in §6.2 to run the variants. We measure the average time used to trigger the vulnerabilities for successful runs. We show the speedup of each variant above the baseline tool, AFLGo, in Table 2. We cannot compute the speedup for several cases when AFLGo

SDFuzz could achieve.

Table 2: Speedup of vulnerability exposure time above AFLGo. We use ✓ to denote the situation that the variant (or SDFuzz) triggers the vulnerability, but the speedup factor is not available because AFLGo does not trigger it.

ID	AFLGo$_{+si}$	AFLGo$_{+et}$	AFLGo$_{+sf}$	AFLGo$_{+df}$	SDFuzz$_{-si}$	SDFuzz$_{bl}$	SDFuzz
1	1.16	1.95	1.27	1.08	2.05	1.34	2.45
2	1.00	2.11	2.11	1.34	2.31	1.42	1.71
3	1.25	1.67	1.00	1.00	1.64	TO	1.67
4	1.30	1.17	1.17	1.08	1.19	1.25	1.36
5	2.21	2.00	0.80	1.60	2.55	1.31	2.67
6	1.24	3.22	1.12	1.16	3.78	2.49	4.68
7	2.10	9.79	3.19	2.49	12.18	TO	18.60
8	2.19	3.58	1.39	1.36	2.39	5.18	7.48
9	2.02	2.42	1.54	1.32	4.29	2.12	5.38
10	✓	✓	TO	TO	✓	TO	✓
11	TO	TO	TO	TO	✓	✓	✓
12	1.96	1.94	1.26	1.16	1.94	1.19	2.44
13	1.09	1.15	1.09	1.14	1.35	TO	1.43
14	1.72	1.97	1.12	1.54	1.78	1.28	3.35
15	TO	TO	TO	TO	TO	TO	✓
16	1.28	1.49	1.07	1.06	2.54	1.24	2.98
17	1.38	4.02	1.94	1.13	3.38	3.71	4.26
18	1.50	1.50	1.39	1.69	2.32	1.98	3.00
19	1.08	1.16	0.78	1.14	3.19	4.29	10.50
20	TO	✓	TO	TO	✓	TO	✓
21	1.60	1.33	1.12	1.04	1.64	2.91	4.00
22	1.06	1.30	1.60	1.00	1.13	1.28	1.46
23	TO	✓	TO	TO	✓	✓	✓
24	6.96	5.66	2.38	2.12	7.49	9.98	12.18
25	1.28	1.90	1.47	1.08	2.58	2.49	4.23
26	1.09	0.95	1.11	1.03	1.13	0.87	1.16
27	1.14	1.19	1.04	1.10	1.03	0.91	1.21
28	2.14	2.73	1.31	1.25	2.41	2.13	3.01
29	TO	TO	TO	TO	TO	✓	✓
30	1.41	1.33	1.31	1.26	1.54	1.92	2.21
31	1.43	1.11	1.24	1.34	1.09	1.21	1.52
32	2.81	4.21	1.71	1.32	4.43	3.37	5.23
33	2.07	1.83	1.29	1.48	1.95	1.87	2.60
34	1.54	1.96	1.20	1.29	2.24	2.18	2.34
35	1.34	1.44	1.32	1.15	1.23	1.72	1.74
36	1.23	1.67	1.29	1.16	1.06	1.27	1.96
37	1.86	2.23	1.21	1.42	2.47	2.17	2.62
38	1.78	2.81	1.49	1.19	2.94	1.28	3.83
39	TO	TO	TO	TO	TO	TO	TO
40	1.32	1.41	1.18	1.09	1.52	1.43	1.59
41	1.54	1.98	1.26	1.05	1.24	1.74	2.13
42	1.76	1.53	1.32	1.15	1.92	1.39	2.02
43	✓	✓	✓	TO	✓	TO	✓
44	TO	✓	TO	TO	✓	✓	✓
45	1.42	1.36	1.29	1.13	1.19	TO	1.45
Avg.	1.56	1.94	1.32	1.24	2.11	1.87	2.83

or the variant does not trigger the vulnerability, and we use ✓ to denote them. Generally speaking, the results validate the effectiveness of the four key techniques. We next analyze the results in detail.

Effect of target states. Target states are the indispensable factor for SDFuzz's superior performance. With only the bug location—a reduced target state, we could observe a significant performance decrease in SDFuzz$_{bl}$, compared to the full-fledged SDFuzz. SDFuzz$_{bl}$ exposed 36 vulnerabilities and SDFuzz$_{bl}$ on average achieved a speedup of 1.87× above AFLGo. We found that SDFuzz$_{bl}$ could not expose several vulnerabilities that even AFLGo could do, *e.g.*, #3. This is because by using only the bug location, SDFuzz$_{bl}$ could not precisely terminate executions. Besides, it would sometimes wrongly terminate the executions because it incorrectly determines the (un)recoverable executions. This demonstrates the necessity of using target states in our approach.

Selective instrumentation. AFLGo$_{+si}$ and SDFuzz share the same required code identification step. Therefore, AFLGo$_{+si}$ could eliminate the same proportion of code as SDFuzz. We have shown the results in §6.2 that SDFuzz

could eliminate 48.18% of unrequired functions on average, demonstrating its high effectiveness. On the contrary, SDFuzz$_{-si}$ would still explore the whole code base.

From Table 2, we found AFLGo$_{+si}$ improved AFLGo by exposing two additional vulnerabilities and triggered 38 out of the 45 vulnerabilities in total. AFLGo$_{+si}$ improved the performance of AFLGo, with an average speedup of 1.56× on the 36 vulnerabilities AFLGo exposed. We also found that SieveFuzz [32] had a similar average performance on vulnerability exposure time as AFLGo$_{+si}$ even though AFLGo$_{+si}$ eliminated more unrequired code. SieveFuzz also achieved an average speedup of 1.56× above AFLGo according to the results in Table 1. The reason is two-fold. First, AFLGo$_{+si}$ sacrifices a bit performance for fault tolerance by its instrumentation-based code elimination. Second, SieveFuzz also employs other techniques such as diversity heuristics that AFLGo$_{+si}$ currently does not support. Integrating such techniques would definitely improve our solution.

We further analyze another variant SDFuzz$_{-si}$ with selective instrumentation disabled atop SDFuzz. SDFuzz$_{-si}$ triggered 43 vulnerabilities and achieved an average speedup of 2.11×. This further confirms the benefit of the selective instrumentation technique.

Early termination of executions. The execution termination technique in AFLGo$_{+et}$ is highly effective. In the 24-hour experiments on the 45 vulnerabilities, AFLGo$_{+et}$ early terminated 56.23% of the executions on average. This also confirms that executions that cannot reach the target states are widespread in real-world programs. We could not measure the proportion of terminated executions in Beacon because it is close-sourced. Besides, AFLGo$_{+et}$ also improved the fuzzing throughput by 8.71× on average, compared to AFLGo. In terms of vulnerability exposure, AFLGo$_{+et}$ triggered five more vulnerabilities than AFLGo, exposing a total of 41 vulnerabilities. AFLGo$_{+et}$ achieved an average speedup of 1.94× above AFLGo on the vulnerabilities AFLGo triggered.

Feedback mechanism. The feedback mechanism guides the testing to trigger the vulnerabilities. AFLGo$_{+sf}$ triggered one more vulnerability compared to AFLGo. It also accelerated the vulnerability exposure by 1.32×. The rationale lies in that AFLGo$_{+sf}$ appropriately spares the testing efforts to desired paths specified in target states. However, we notice that AFLGo$_{+sf}$ performed slightly worse than AFLGo in several cases like #5 and #19. The reason mainly came from the runtime overhead caused by the program state maintenance. Since the cases are relatively simple to trigger, the overhead turned to take a significant proportion in the total used time.

Similarly, the precisely-weighted distance metric feedback in AFLGo$_{+df}$ also improved fuzzing effectiveness. Compared to AFLGo, AFLGo$_{+df}$ achieved an average speedup of 1.24×. However, we notice that the performance improvement of AFLGo$_{+df}$ is slightly less significant than that of AFLGo$_{+sf}$. Our manual analysis found that the target state feedback could offer more benefits in optimizing the exploration direction compared to the new distance metric.

Table 3: Vulnerability discovery results.

Program	Statically Reported	SDFuzz Validated
libjpeg	46	2
tinyexr	22	1
pugixml	59	1
ffmpeg	32	0
Total	159	4

6.5 New Vulnerability Discovery

We further evaluate the efficacy of SDFuzz in discovering new vulnerabilities. In particular, we explore the feasibility of applying SDFuzz to automatically validate the analysis results of SVF [35]. We apply SVF to a set of well-tested applications that handle or process different types of files, such as libjpeg [37], tinyexr [36], pugixml [42], and ffmpeg [8]. Some of the applications have been well-tested by the related tools [7, 26].

We employed the saber checker [35] of SVF to identify memory leakage and double-free vulnerabilities. In total, SVF reported 159 suspicious cases, and we leveraged SDFuzz to validate them. SVF provided a program flow for each suspicious case, which SDFuzz converted to a target state for directed fuzzing. Given the large number of cases, we ran SDFuzz for 12 hours for each case, resulting in a total CPU time of around 2,000 hours. To date, SDFuzz successfully identified four new vulnerabilities. We have responsibly reported the new vulnerabilities to the vendors and are in the process of applying for new CVE IDs. Three of the vulnerabilities have been acknowledged. The detailed vulnerabilities can be found in Table 3. Given this, we believe that SDFuzz can be applied in practice as a fully automated solution for vulnerability validation.

To understand the efficacy of target states, we further replayed the crashing inputs generated by SDFuzz on the four vulnerabilities and analyzed the triggered program paths. Our investigation revealed that the four vulnerabilities were triggered through the exact paths reported by SVF (*i.e.*, paths derived from the target states). This confirmed that the target states could help validate the static analysis results.

7 Discussion and Limitations

In this section, we discuss the limitations of SDFuzz and future work.

Requirement of target states. SDFuzz requires target states from two sources—vulnerability reports and static analysis results. This requirement can be satisfied in practice as most of the vulnerability reports provide such target states (§3).

Even if vulnerability reports are not available, SDFuzz can still leverage existing static bug analysis techniques to identify the target states. Compared to exploring all paths, the paths in static analysis results often convey the suspicious flows that better deserve testing. We have already demonstrated the feasibility in §6.5.

We admit that some application scenarios such as patch testing might not provide available target states. Our solution can extract the target states from the vulnerability where the patch applies by analyzing the vulnerability report. It can also analyze the patch change logs to identify the target reaching order [16]. However, it currently cannot directly derive complete target states from only the patch change logs, *e.g.*, find the problematic program paths in the patches. To mitigate this problem, one possible direction is to employ advanced program analysis techniques such as symbolic execution [2,5] to synthesize feasible target states for a patch. We leave this as future work.

States other than target states. A downside of our approach is that it might potentially overlook some valuable paths that are not included in the target states, only in crash reproduction. In fact, as shown in our evaluation, SDFuzz could trigger the vulnerabilities in a significantly shorter time. This suggests that overlooking other states would not undermine the performance of SDFuzz.

We believe driving to target states is a reasonable trade-off for two reasons. First, infeasible paths for triggering the vulnerabilities dominate the program paths [43]. The paths stated in the target states are preferred working ones. Our solution would mostly exclude unnecessary exploration and guide the fuzzer towards these guaranteed working directions. Second, the paths in target states are likely to be the best or simplest ones to trigger the vulnerability as they correspond to the first cases when a vulnerability gets triggered or reported in real world. Other fuzzers ultimately triggered the vulnerabilities also through the paths in the target states. This confirms the rationale of using target states.

In the application scenario of validating static analysis results, the task of applying directed fuzzing is to test the reported suspicious flows. Therefore, our strategy of focusing on the target states (suspicious flows) is reasonable for achieving this goal.

Incomplete call graph. We notice that the call graph might not be complete because some call targets cannot be statically inferred using SVF. SDFuzz might incorrectly assess the state recovery capability because of wrong path existence judgment. To mitigate the issue, we can incorporate dynamic tracing to monitor function invocations, and accordingly add additional call edges to the call graph. We can also leverage other advanced type inference methods to refine the call graph [18, 19].

8 Related Work

Eliminating unnecessary explorations can improve fuzzing effectiveness. Beacon [14], SieveFuzz [32], and SelectFuzz [20] are three typical examples considering the reachability to the target sites. As we discussed in §2.3, SDFuzz can further exclude unnecessary reachable executions that cannot reach the target states.

The distance metric is also an important factor in directed fuzzing. AFLGo [26] and SemFuzz [41] were the first lines of research about directed fuzzing. They initially proposed the concept of distance metrics to drive the coverage-guided fuzzing towards a direction. Hawkeye [3] further used adjacent-function distance, which considered the number of times a function is called. ParmeSan [24] directed the testing towards locations with more sanitization checks. WindRanger [7] pointed out that only deviation basic blocks were necessary for distance computation. MC2 [30] approached a new randomized search algorithm to optimize the fuzzing exploration. SDFuzz differentiates itself from these tools by considering the target states for triggering vulnerabilities. Additionally, LOLLY [17] analyzed the execution trace of a test case after its execution and used that as feedback. Its analysis was purely offline, *i.e.*, it did not capture the runtime program behaviors like the calling contexts. It also introduced heavy overhead in recording the complete execution trace in every fuzzing trial. SDFuzz instead selectively monitors the runtime program states with minimal overhead. CAFL [16] measured the order dependency among targets—a part of the target states. SDFuzz considers a comprehensive set of the target states for the feedback.

Unlike SDFuzz, some related works improve DGF from other angles or apply DGF to other scenarios. FuzzGuard [43] used deep learning to predict unreachable test cases and filtered them out from the testing. As for binary programs, UAFuzz [23] considered the target reaching order and detected UAF vulnerabilities; 1dVul [25] analyzed binary patches to identify one-day vulnerabilities.

9 Conclusion

Directed grey-box fuzzing would often unnecessarily explore code and paths that cannot trigger the vulnerabilities. In this paper, we presented SDFuzz, an effective directed fuzzer driven by target states to mitigate this problem. SDFuzz excludes unnecessary explorations by eliminating unrequired code and early terminating executions that cannot reach the target states. SDFuzz further employs a two-dimensional feedback mechanism to proactively guide the testing direction. Our evaluation results demonstrated that SDFuzz could trigger the vulnerabilities faster and outperformed the prior works. SDFuzz also discovered four previously unknown vulnerabilities, proving its practical value in automated vulnerability validation.

References

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[3] Hongxu Chen, Bihuan Chen, Yinxing Xue, Xiaofei Xie, Yang Liu, Yuekang Li, and Xiuheng Wu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[4] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering*, 2014.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, March 2011.

[6] Ren Ding, Hong Hu, Wen Xu, and Taesoo Kim. Desensitization: Privacy-aware and attack-preserving crash report. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2020.

[7] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, May 2022.

[8] ffmpeg. A complete, cross-platform solution to record, convert and stream audio and video, 2022. https://ffmpeg.org/.

[9] Github. Codeql, 2023. https://codeql.github.com/.

[10] GNU. Backtraces, 2022. https://www.gnu.org/software/libc/manual/html_node/Backtraces.html.

[11] Google. Crash-reporting system., 2022. https://chromium.googlesource.com/breakpad/breakpad.

[12] Google. Google fuzzer test suite, 2023. https://github.com/google/fuzzer-test-suite.

[13] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[14] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[15] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, June 2012.

[16] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual event, August 2021.

[17] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019.

[18] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2023.

[19] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[20] Changhua Luo, Wei Meng, and Penghui Li. SelectFuzz: Efficient directed fuzzing with selective path exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2023.

[21] Magma. A ground-truth fuzzing benchmark suite based on real programs with real bugs., 2023. https://hexhive.epfl.ch/magma/docs/bugs.html.

[22] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, 2010.

[23] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, San Sebastian, Spain, October 2020.

[24] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. *parmesan*: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.

[25] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *Proceedings of the 2019 International Conference on Dependable Systems and Networks (DSN)*, Portland, Oregon, June 2019.

[26] Marcel Pham, Van Thuan, Manh Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[27] Marcel Pham, Van Thuan, Manh Dung Nguyen, and Abhik Roychoudhury. Github repository of aflgo, 2022. https://github.com/aflgo/aflgo.

[28] Korosh Koochekian Sabor, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj. Automatic prediction of the severity of bugs using stack traces. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, 2016.

[29] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

[30] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, November 2022.

[31] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[32] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction. In *Annual Computer Security Applications Conference*, Austin, TX, 2022.

[33] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, November 2016.

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

[35] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 2014.

[36] syoyo. Tiny openexr image library, 2023. https://github.com/syoyo/tinyexr.

[37] thorfdbg. libjpeg, 2023. https://github.com/thorfdbg/libjpeg.

[38] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[39] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[40] yguoaz/. Docker of beacon, 2022. https://hub.docker.com/r/yguoaz/beacon.

[41] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[42] zeux. Light-weight, simple and fast xml parser for c++ with xpath support, 2023. https://github.com/zeux/pugixml.

[43] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.