DAFL: Directed Grey-box Fuzzing guided by Data Dependency
Tae Eun Kim, KAIST; Jaeseung Choi, Sogang University; Kihong Heo and Sang Kil Cha, KAIST
https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.
August 9–11, 2023 • Anaheim, CA, USA
978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.
DAFL: Directed Grey-box Fuzzing Guided by Data Dependency

Tae Eun Kim
KAIST

Jaeseung Choi
Sogang University

Kihong Heo
KAIST

Sang Kil Cha
KAIST

Abstract
Despite growing research interest, existing directed grey-box fuzzers do not scale well with program complexity. In this paper, we identify two major scalability challenges for current directed grey-box fuzzing. Particularly, we find that traditional coverage feedback does not always provide meaningful guidance for reaching the target program point(s), and the existing seed distance mechanism does not operate well with programs with complex control structures. To address these problems, we present a novel fuzzer, named DAFL. DAFL selects code parts that are relevant to the target location and obtains coverage feedback only from those parts. Furthermore, it computes precise seed distances considering the data-flow semantics of program executions. The results are promising. Out of 41 real-world bugs, DAFL was able to find 4, 6, 9, and 5 more bugs within the given time, compared to AFL, AFLGo, WindRanger, and Beacon, respectively. Furthermore, among the cases where all fuzzers produced a median TTE, DAFL was at least 4.99 times faster on average compared to 3 state-of-the-art directed fuzzers including AFLGo, WindRanger, and Beacon.

1 Introduction
Directed Grey-box Fuzzing (DGF) has been gaining momentum in software security due to its ability to generate reproducible test cases from bug reports. For instance, static analysis tools often report buggy lines of code while not providing concrete test cases for the bugs found [3–5, 7, 12, 22, 31, 51]. Thus, DGF can help prove the validity of such reported bugs.

Current DGF is mainly based on two key mechanisms: 1) DGF evolves test cases by favoring ones that cover new execution paths as in traditional (undirected) grey-box fuzzing; 2) DGF provides guidance to fuzzers in reaching the target program point(s) by weighing the precedence of each test case depending on the distance between the exercised nodes and the target node(s) in the Control-Flow Graph (CFG).

Unfortunately, these mechanisms are subject to critical challenges, which can make DGF no better or even worse than undirected fuzzing when handling programs with complex control structures.

Challenge 1. Code coverage can give negative feedback to DGF. As long as fuzzers can achieve more coverage, they can be directed to paths that are irrelevant to the target execution. This problem becomes worse when the target program is large as it can include more irrelevant code parts. To our knowledge, Beacon [32] is the first in mitigating this problem by essentially preventing irrelevant executions. Specifically, it computes the weakest precondition for reaching the target and modifies the program to early-terminate executions when the precondition is not met. However, such a technique does not scale with complex programs because precisely obtaining weakest preconditions from a real-world program is infeasible. Indeed, our preliminary study shows that programs with complicated loops will effectively disable Beacon’s weakest precondition analysis (see §2.1).

Challenge 2. Current distance-based feedback mechanisms do not operate well with programs with complex control structures. Most existing directed grey-box fuzzers—such as AFLGo [9], Hawkeye [15], and Beacon [32]—compute a seed distance, i.e., a precedence score, of a test case, by considering all the executed nodes in the CFG. However, their distance mechanisms can give wrong guidance toward irrelevant nodes, especially when the program is large because long execution paths are likely to include many nodes that are semantically irrelevant to the target. In such cases, existing seed distance mechanisms can either prefer or penalize an input based on the irrelevant part of the execution. To our knowledge, WindRanger [23] is the first in tackling the bias issue by selectively considering critical nodes named Deviation Basic Blocks (DBBs). However, DBBs can be a bad representative when they are within a loop as we will show in §2.2.

Our Solution. In this paper, we present two novel techniques for DGF to tackle the two aforementioned problems. First, selective coverage instrumentation reduces negative feedback by collecting code coverage information only from those code parts relevant to the target execution. Ours is dif-
different from the path-pruning technique of Beacon in that we do not modify the target program, nor remove any program parts. Instead, selective coverage instrumentation simply skips adding instrumentation to those irrelevant program parts so as to not collect coverage information from them. This design choice helps relax the soundness requirement of our analysis, unlike Beacon, making ours applicable to more complex programs with loops.

Second, semantic relevance scoring provides a precise way to compute seed distance more intuitively compared to the existing methods. The primary intuition here is that complex control structures, such as loops, in a CFG can often be eliminated when considering a Def-Use Graph (DUG). Therefore, instead of computing seed distances from a CFG, we propose to compute seed distances from a DUG. Our empirical study shows that such design choice effectively guides fuzzing toward target bugs while reducing analysis time.

We present DAFL, a new directed grey-box fuzzer that implements our techniques to tackle both of the challenges. Our experimental results show substantial improvements over existing fuzzers. Particularly, we evaluated DAFL on 41 real-world bugs used in previous literature, and showed that DAFL was able to reproduce the bugs significantly faster than existing directed fuzzers.

Finally, it is important to note that not many DGF tools are readily available. Even if a tool itself is available, the associated utility, such as a triage script, is not. Hence, reproducing previous results is extremely difficult if not impossible. We have communicated with the authors of the existing papers and put significant effort to make fair comparisons. Thus, we truly believe one of our contributions is to fully publicize our toolchain as well as our dataset.

In summary, our contributions are:

- We present selective coverage instrumentation, a novel way to measure code coverage for DGF.
- We devise semantic relevance scoring, a novel technique to effectively schedule seeds for DGF.
- We design and implement DAFL incorporating both techniques.
- We publicize DAFL to support open science via Zenodo and GitHub:
  
  https://doi.org/10.5281/zenodo.8031029 and
  https://github.com/prosyslab/DAFL-artifact

2 Motivation

We motivate our approach by considering a buffer overflow bug found in swftophp of the libming project [1], which is assigned CVE-2017-7578 [59]. Figure 1 shows a simplified code snippet. The program first opens a file (Line 9) and reads a stream of input data from the file by iterating the for-loop.

When the published Proof-Of-Concept (PoC) [59] is given as input, the program assigns integer 46 to variable type (Line 14), which is used to indirectly call the parser function parseSWF_DEFINEMORPHSHAPE (Line 30). The parser function then calls the vulnerable function parseSWF_RGBA to save the color information in the GradientRecords array (Line 37). Note that the size of the array is 8 (Line 35), while the index variable i can have an arbitrarily large number depending on the user input (Line 36). Thus, this program may crash at Line 42 due to invalid memory access. The goal of this paper is to guide a fuzzer to automatically generate such inputs (e.g., type and i) for discovering the target bug (e.g., Line 42).

2.1 Challenge 1: Negative Feedback

DGF selects test cases that can achieve new code coverage, but this strategy can provide negative feedback in reaching the target location. In this example, there are 389 functions reachable from the main function. However, only 31 of them
are exercised with the PoC. Figure 2 shows a simplified call graph where the shaded nodes represent functions executed when the PoC is used as input. While blockParse can call more than 80 different functions depending on the input, only one function, i.e., parseSWF\_DEFINEMORPHSHAPE, is relevant to the target bug. Nevertheless, existing directed fuzzers assign a good amount of energy to the test cases that explore such new (but irrelevant) functions, thereby significantly misguiding them.

Beacon [32] mitigates this problem by pruning infeasible paths that cannot reach the target, but it is insufficient to handle complex programs such as this example. Ideally, Beacon should have generated the weakest precondition for reaching the target bug. For example, it could have found that the value of type (in Line 14 and 29) should be 46, and could have created a statement asserting the precondition (i.e., type == 46). By putting the assert statement right after Line 14, Beacon could have prevented irrelevant executions in an ideal scenario. In practice, however, Beacon fails to infer any preconditions from the functions main and blockParse due to their complex control structures. Note that inferring weakest preconditions is challenging in the presence of complicated loops. Consequently, Beacon cannot provide any guidance to the target point in this example. This suggests the need for a scalable way to eliminate negative feedback (§3.1).

2.2 Challenge 2: Misleading Distance Metrics

Current DGF guides the search by prioritizing test cases based on their syntactic distances to the target on the control-flow graphs, but such a syntactic metric often fails to reflect the semantic aspects of the target bug.

Suppose there are two seed inputs \( s_A \) and \( s_B \) that explore different paths in the example program. Figure 3a depicts the executions where each node corresponds to a line number in Figure 1. \( s_A \) is a malformed file, and the execution with \( s_A \) takes the true branch of the first conditional (Line 12) and quickly terminates the program. AFLGo computes the seed distance of \( s_A \) by taking the average of the distances between the executed nodes and the target node, which results in 34.5 in our example. On the other hand, seed \( s_B \) is a well-formed file, and thus, is more relevant to the target bug. The execution with \( s_B \) passes the first sanity check in Line 12, although it does not pass the second sanity check in Line 17. While \( s_B \) is semantically more relevant to the bug, AFLGo assigns a higher seed distance 36.75 to seed \( s_B \) than \( s_A \), meaning that AFLGo will prefer \( s_A \) over \( s_B \).

Other state-of-the-art approaches fundamentally have the same limitation. Notably, WindRanger [23] computes the average of the distances between a subset of executed nodes and the target node. It selects the subset, named deviation basic blocks (DBB), by considering nodes in the CFGs that deviate the execution from the target location. In our example, the deviation basic block of \( s_A \) is block 12 in Figure 3b, because the block deviates the execution from the loop. Thus, WindRanger gets 34 as its seed distance. Similarly, WindRanger computes the seed distance of \( s_B \) as 45 (considering its deviation basic block as block 17) which is higher than that of \( s_A \). Notice that WindRanger still has the same problem as AFLGo: the DBB-based distance metric can also mislead the guidance when complicated control structures are involved such as loops. This motivates our semantic relevance scoring, a semantically intuitive and precise distance feedback mechanism (§3.2).

3 Overview

This section gives a high-level overview of DAFL, which tackles the two challenges. DAFL employs two major techniques to handle each challenge: 1) selective coverage instrumentation, and 2) semantic relevance scoring.

Figure 4 illustrates the overall architecture of DAFL, which consists of two major phases: static analysis and fuzzing phase. During the static analysis phase, DAFL takes in as input a program with an annotated target location and runs an inter-procedural static analysis to identify all the statements that the target location is data-dependent on. Our analysis returns a Def-Use Graph (DUG) and a set of relevant functions with respect to the target point. DAFL then performs selective coverage instrumentation (§3.1), which instruments only the relevant functions in the target program. This enables DAFL to selectively receive coverage feedback only from the dependent parts of the program during the next fuzzing phase.

In the fuzzing phase, DAFL iteratively runs the instrumented program as in traditional grey-box fuzzing, but it gets code coverage only from the relevant functions due to selective coverage instrumentation. Furthermore, DAFL employs a novel scheduling algorithm that we call semantic relevance scoring (§3.2), which prioritizes seeds with respect to their
relevance scores, which are derived from the DUG, rather than syntactic distances.

Under the guidance of the two techniques, DAFL significantly outperforms existing fuzzers. Specifically, we ran DAFL, AFL, AFLGo, Beacon, and WindRanger, and measured time to generate a test case that triggers CVE-2017-7578 shown in the example. As a result, DAFL, AFL, AFLGo, and Beacon were able to discover a buggy test case in 139s, 283s, 636s, and 1,058s, respectively (WindRanger is ignored because it resulted in a timeout after 24 hours). That is, DAFL was at least 2 times and at most 7.6 times faster than the state-of-the-art tools in exposing this bug.

### 3.1 Selective Coverage Instrumentation

Unlike existing directed fuzzers, DAFL selectively receives coverage feedback only from a portion of the target program that is relevant to the target location. To identify such relevant parts, we traverse the program backward starting from the target location on the DUG, and collect all dependent statements. In the example, we collect the nodes shown in Figure 3c.

While this set of nodes contains semantically relevant parts to the target location, data dependencies may miss critical conditional statements that determine the reachability to the target location (i.e., control dependency). For example, the conditional statement in Line 29 is crucial to reaching the target point but is not a part of data dependency. However, collecting all the data- and control-dependent nodes transitively can result in a significant precision loss.

To address this issue, we collect all functions in the sliced DUG and instrument all the nodes in the functions to receive coverage feedback. This in turn includes control-dependent nodes that are closely related to the data dependencies, while preventing excessive overapproximation. In Figure 1, our approach selects only 4 functions out of 389. The identified functions are shown as shaded nodes in Figure 2. This selective instrumentation can guide the fuzzer toward the relevant functions (e.g., `parseSWF_DEFINEMORPHSHAPE`) without be-
### 3.2 Semantic Relevance Scoring

We evaluate seed inputs by their semantic relevance rather than the syntactic distances. We define a score of a seed input as the sum of semantic relevance scores of executed nodes in the DUG, where the score of each node is defined based on the proximity to the target point. Specifically, we assign score 1 to the farthest nodes and the maximum score, i.e., the distance from the farthest node to the target, to the closest nodes from the target. For example, nodes 9 and 14 have score 1 in Figure 3c as they are the farthest nodes. On the other hand, node 37 is assigned 4 which is the distance of the farthest nodes 9 and 14. The score of the target node is simply assigned 5. Then, the semantic relevance score of seed $s_A$ in Figure 3c which executes only Line 9 is 1. On the other hand, the score of seed $s_B$ is assigned 2 as it executes two nodes 9 and 14 appear in the DUG. This in turn leads DAFL to prioritize $s_B$ over $s_A$ during a fuzzing campaign, thereby addressing Challenge 2.

## 4 Design

In this section, we detail the design of DAFL, which is outlined in Algorithm 1. At a high level, DAFL is a function that takes in as input a program $P$ and a target program point $t$, and returns a set of crashing test cases found.

DAFL first statically analyzes the data dependency of $P$ and slices $P$ with regard to $t$ using the data dependency (Line 1). The function Slice returns a tuple $(G,F)$: 1) $G$ is a DUG obtained by following the data definitions from $t$, and 2) $F$ is a set of functions that are covered by $G$. We use the thin slicing technique [57] to focus on value transfers, as we will further detail in §4.1. The set of sliced functions $F$ serves as a basis for selective coverage instrumentation, which essentially instruments only those functions in $F$ and returns an instrumented program $P'$.

DAFL then enters the fuzzing phase, where it first initializes both the seed pool and the set of crashing test cases (Line 3–4). Next, DAFL iterates the while loop to generate test case(s) that can reach $t$. For each iteration, DAFL chooses a seed $s$ from the seed pool (Line 6) and assigns energy to $s$ based on the semantic relevance score, which essentially decides how many mutants to make from the seed (Line 7). For each mutant test case $s'$ generated from $s$, DAFL then runs $P$ with $s'$ while selectively measuring coverage only from the sliced functions $F$. For every mutant execution, DAFL computes the semantic relevance score for each mutant using $G$ (Line 11). We further detail our seed scheduling strategy in §4.2.

### 4.1 Program Slicing

Traditional program slicing computes a set of all possible source lines that may affect a given program point. However, we note that it can produce an unnecessarily large slice as output. Consider the following example:

```plaintext
1  x = f();
2  y = g();
3  p = δy;
4  z = x + *p;
```

Let us assume that our goal here is to compute a static program slice for $z$ in Line 4. In our setup, this means Line 4 is our target line to reach. Since the variable $z$ in Line 4 is affected by the variables $x$, $y$, and $p$, a naive slicing algorithm will return Line 1–3 as a result. However, we note that Line 3 merely gets an address of the variable $y$, and the actual value copies are performed through Line 1 and 2. That is, we can potentially obtain a compact slice from the example program, which contains only Line 1 and 2.

Thin slicing [57] addresses this challenge by ignoring data dependencies through pointer dereferences. In this example, the result of thin slicing concisely includes the definition of two variables that were directly used as the addition operands in Line 4. Consequently, thin slicing returns Line 1–2 as a result in the above example. As we will empirically show in §5.3, thin slicing provides more effective guidance for directed fuzzing.

Unlike typical applications of slicing (e.g., debugging or program understanding), fuzzing has to actually execute control-dependent nodes of the target location even though they are not in the thin slice. However, as briefly discussed in §3.1, it is impractical to collect all data- and control-dependent nodes because of the over-approximation.

---

**Algorithm 1: DAFL (P,t)**

1. $G,F \leftarrow \text{Slice}(P,t)$ \hspace{1em} // §4.1
2. $P' \leftarrow \text{SelectiveCovInstr}(P,F)$ \hspace{1em} // §3.1
3. $Pool \leftarrow \text{InitializeSeedPool}()$
4. $Crashes \leftarrow \emptyset$
5. while not timeout do
   6. \hspace{1em} $x,scr \leftarrow \text{Choose}(Pool)$ \hspace{1em} // §4.2
   7. \hspace{2em} $e \leftarrow \text{AssignEnergy}(scr)$ \hspace{1em} // §4.2
   8. for $e$ times do
      9. \hspace{3em} $s' \leftarrow \text{Mutate}(s)$
      10. \hspace{4em} $cov \leftarrow \text{MeasureCov}(P',s',F)$
      11. \hspace{4em} $scr \leftarrow \text{ComputeScore}(P',s',G)$ \hspace{1em} // §4.2
      12. \hspace{5em} if cov has any gain then
         13. \hspace{6em} Pool $\leftarrow \text{Add}(Pool,s',scr)$
         14. \hspace{6em} if $s'$ crashes $P'$ then
            15. \hspace{7em} $Crashes \leftarrow Crashes \cup \{s'\}$
   16. return $Crashes$
Without this information, we cannot figure out which variables where
between exercised nodes and a target node in the sliced DUG.

Instead, we observed that most of the important control-
dependencies appear lexically near the thin slice nodes as also discussed in the original paper [57]. Thus, we instru-
ment functions that include sliced nodes, while computing
relevance scores using the sliced nodes, as will be discussed in
section 4.2. In our experience, this two-level scheme en-
ables DAFL to effectively explore code blocks near the target
location while prioritizing semantically relevant seeds.

We now formally describe our slicing module. Given a
program $P$ and a target point, our slicing algorithm computes a
DUG that contains data-flows related to the provided target
point. We assume that the program is represented as a CFG
$(C, \rightarrow)$ where $C$ is the set of program points and $(\rightarrow) \subseteq C \times C$
is the set of (inter-procedural) control flow edges. A target
location $t \in C$ is a node in the control flow graph, and our
goal is to generate a test case that reaches $t$. The DUG $(C, \rightarrow)$
of $P$ has the same set of nodes as the CFG, but it comprises
data dependency edges $(\rightarrow)$ rather than control flow edges $(\rightarrow)$.
We define the data dependency edges based on the thin slicing strategy [57]:

$$c_1 \rightarrow c_2 \iff c_1 \rightarrow^+ c_2 \land \text{x is defined at } c_1 \land
x \text{ is used at } c_2, \text{ but not for pointer dereference } \land
x \text{ is not defined at any node between } c_1 \text{ and } c_2,$$

where $x$ is a variable in $P$.

Note that even with thin slicing, it is still necessary to
know which variables are pointed to by each pointer variable.
Without this information, we cannot figure out which variables
are defined and used in each program point. Therefore, we
perform a flow-insensitive and context-insensitive pointer
analysis to derive the data dependencies.

Once the whole DUG is obtained, we prune away the DUG
to obtain $G$ whose nodes are relevant to the target location $t$.
Particularly, we traverse the whole graph backward starting
from the target node and construct the sliced one. Formally,
given the whole graph $(C, \rightarrow)$ and the target node $t$, we con-
struct the sliced graph $G = (C_t, \rightarrow_t)$ with respect to $t$ where

$$C_t = \{c \in C \mid c \rightarrow^+ t\}$$

$$\rightarrow_t = \{(c_1, c_2) \mid c_1 \in C_t \land c_2 \in C_t \land c_1 \rightarrow c_2\}.$$

Finally, we compute the set of relevant functions by collecting
all the functions involved in the sliced graph:

$$F = \{f \mid f \text{ is the function that contains } c \land c \in C_t\}.$$

4.2 Seed Scheduling

Recall from §2.2 that the current DGF relies on a seed distance
mechanism, which does not reliably represent the semantic
distances between two executions. Our seed scheduling in-
stead leverages a novel feedback mechanism that we call
semantic relevance score, which measures the proximity be-
tween exercised nodes and a target node in the sliced DUG.

By using a sliced DUG over a CFG for calculating the
feedback, we gain two major advantages. First, we can nat-
urally disregard irrelevant nodes as the sliced DUG simply
does not include them. Second, even if a CFG has loop(s), the
corresponding DUG will not have a loop as long as there is no
cyclic data dependency as we showed in Figure 3. Therefore,
it is unlikely to make a miscalculation as in the cases shown in
§2.2, where a semantically closer execution has a longer seed
distance.

4.2.1 Semantic Relevance Score

DAFL computes the semantic relevance score for each seed
input defined with the sliced DUG $G$ obtained from the static
analysis phase. We give a higher score to a seed if 1) the
program execution with the seed exercises more nodes in $G$, or
2) the program execution with the seed exercises nodes
that are closer to the target point. Intuitively, the semantic
relevance score reflects how relevant the seed is to the target
point in terms of data dependency.

More formally, we define the relevance score of each node
based on the distance to the target node in the sliced DUG.
Let $|c_1 - c_2|$ denote the shortest distance between two nodes
$c_1$ and $c_2$ in a DUG. Given a DUG $G$ and the target node $t$,
the semantic relevance score of a node $c$ is:

$$\text{Score}_{G,t}(c) = L - (|c - t|) + 1$$

where $L = \max_{c \in C} |c - t|$ is the distance between $t$ and its
farthest node on $G$. In other words, the farthest node from the
target point is assigned the lowest weight, which is one, and
the closest node is assigned the highest weight.

We now define the semantic relevance score of a seed input,
which is obtained by accumulating the relevance scores of the
nodes in the DUG. Particularly, we observe which nodes in
the DUG is exercised during the course of execution of the
program with the given seed input, and sum all the relevance
scores of exercised nodes. Let $C_s$ be the set of nodes in $G$
exeuted with seed $s$. By abuse of notation, the score of seed
$s$ is defined as the sum of the scores of the covered nodes:

$$\text{Score}_{G,s}(s) = \sum_{c \in C_s} \text{Score}_{G,t}(c).$$

4.2.2 Seed Pool Management

The first use of our semantic relevance score is to prioritize
the seeds for fuzzing. Following the design of AFL [63], we
basically maintain the seed pool as a circular queue and
choose seeds in sequence. However, we prioritize the seeds
based on the relevance score rather than the distance over
control-flow graphs. Whenever a new seed is added (Line 13
of Algorithm 1), all the elements in the pool are sorted by the
relevance score. This enables DAFL to select more promising
seeds earlier during the fuzzing campaign. Notice that the
4.2.3 Energy Assignment

The semantic relevance score also determines how much time to spend fuzzing each seed. In line 7 of Algorithm 1, we decide the number of mutants to derive from the chosen seed. This is often referred to as energy in the literature [9, 10, 63].

To derive more mutants from promising seeds, we assign more energy to a seed input with a higher relevance score. While basically following the same energy assignment scheme as the existing work [9, 10, 63], we further adjust the scheme by using the relevance score of each seed. The conventional energy assignment scheme $E_{AFL}(s)$ is defined by various factors of the seed $s$, such as the length of the execution trace and its execution speed. In addition to that, we multiply the following factor by $E_{AFL}(s)$:

$$E_{DAFL}(s) = \frac{scr_s}{scr_{avg}} \times E_{AFL}(s)$$

where $scr_s$ is the semantic relevance score of the seed $s$ and $scr_{avg}$ is the average score of all the seeds in the seed pool.

4.3 Implementation

We implemented DAFL on top of AFL [63] v2.57b. Specifically, we added 36 lines of C++ code in the AFL’s LLVM pass for selective coverage instrumentation in order to skip instrumenting program locations filtered out from the slicing step. We also added 489 lines and deleted 195 lines of C and C++ code in the AFL’s LLVM pass and the scheduling algorithm. In total, DAFL consists of 8,902 SLoC of C and C++ code. The program slicing module is implemented on top of SPARROW [46], a static analysis framework for C programs. We added 2,150 SLoC and deleted 415 SLoC of OCaml code for implementing the slicing module. We make our source code publicly available at Zenodo And GitHub: https://doi.org/10.5281/zenodo.8031029 and https://github.com/prosyslab/DAFL-artifact

5 Evaluation

This section answers the following research questions:

**RQ1.** How fast is DAFL in terms of reproducing target bugs?

**RQ2.** How does the choice of slicing strategies affect the performance of DAFL?

**RQ3.** How do our techniques (selective coverage instrumentation and semantic relevance scoring) affect the performance of the fuzzing results?

5.1 Evaluation Setup

**Baselines.** We compare DAFL with the following four state-of-the-art fuzzers that are publicly available:

- AFL [63]: v2.57b.
- AFLGo [9]: commit b170f9d.
- Beacon [32]: Docker SHA256 hash a09c8cb.
- WindRanger [23]: Docker SHA256 hash 8614ceb.

We excluded Hawkeye [15] and ParmeSan [47] from our comparison for the following reasons. Hawkeye is not publicly available. ParmeSan is public, but we had trouble reproducing the results reported in the original paper. We note that Herrera et al. [29] also reported the same problem.

**Benchmark.** We evaluated the performance of the fuzzers using various types of security vulnerabilities in C programs. Specifically, we used 41 vulnerabilities from the Beacon [32] paper. Among all the vulnerabilities in Beacon’s benchmark, we excluded 14 vulnerabilities because 1) 7 of them were from C++ programs while our static analyzer only supports C, 2) bug reports for 5 of the vulnerabilities were not accessible, as the database is not maintained anymore, and 3) none of the fuzzers were able to reproduce 2 of the vulnerabilities within 24 hours. These two vulnerabilities require very specific inputs to trigger the bugs. For example, CVE-2018-13785 is caused by a division-by-zero that is triggered only when the width and channel of the input PNG file are set to 0x55555555 and 0x3, respectively. Therefore, fuzzers have extremely low chances to reproduce these vulnerabilities. The chosen vulnerabilities are shown in Table 1. We highlight that this benchmark includes 6 CVEs from Binutils which are commonly used in other directed fuzzing papers [9, 15, 23, 47] as well.

Note that we built all the target binaries using ASAN [54] in the main experiment, since it is a common practice in most fuzzing campaigns that aim to expose vulnerabilities. However, Beacon does not support fuzzing a target binary compiled with ASAN. Therefore, we ran a supplementary experiment without ASAN to compare DAFL with Beacon.

**Evaluation Criteria.** To evaluate the performance of directed fuzzers in a crash reproducing test, we first have to set precise criteria to specify a target bug, i.e., target program location(s), and to decide whether the bug is found or not. Unfortunately, none of our baseline fuzzers provides their criteria, which makes it extremely difficult to have the same setting as the ones used in their original experiments. We, hence, carefully set our criteria after multiple communications with the authors of the previous work, and we will publicize our evaluation criteria for follow-up research.

Specifically, we obtained the target location of each bug by running the ASAN-enabled program with the PoC input stored in the MITRE CVE database [45]. We then chose our
target location from the stack trace obtained from an ASAN report.

To determine whether the targeted bug is found by a crashing input, we first replayed the input with the ASAN-enabled program. We then examined the generated ASAN report to triage the crash with the following three criteria. First, we regard a crash as the targeted one if the crash type and the crash line are identical to those of the PoC. Second, for the cases where different bugs of a program can have the same crash location (e.g., CVE-2017-11728 & CVE-2017-11729), we additionally checked if the immediate caller of the crash line matches. Finally, for the stack overflow cases (i.e., infinite recursion), we checked whether all the functions that appear on the stack trace of the PoC are included in the stack trace of our crashing input.

Recall that ASAN is not used in the fuzzing stage of the supplementary experiment against Beacon. While ASAN was not used during the fuzzing, we replayed the found crashes with the ASAN-enabled program afterwards to obtain the ASAN report. Based on this report, we could use the same triaging criteria in both the main experiment and the supplementary experiment against Beacon.

To evaluate the fuzzers, we measured the Time To Exposure (TTE) for each bug. For TTE, we consider not only the running time of fuzzing, but also that of static analysis if the fuzzer uses static analysis to obtain information from the program to guide the fuzzing (i.e., DAFL, AFLGo, and Beacon). This is because static analysis time is often not a one-time cost in practice. For example, in the setting of continuous fuzzing (i.e., targeting changed code) the analysis must be performed every time the code changes. Unless otherwise noted, we used the same metric for all experiments.

All tools were given 24 hours of time budget for the fuzzing step. We repeated the experiments 40 times and took the median value to minimize the impact of the fluctuation due to the innate randomness in fuzzing.

Environment. We ran the experiment on the machines equipped with Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz with 64 cores using Ubuntu 20.04 LTS. Each fuzzing session was run on a Docker container assigned with a single CPU core and 4GB of memory. We utilized only 40 out of 64 CPU cores, all running the same fuzzing session.

5.2 Time-to-Exposure

We first evaluate the effectiveness of DAFL in terms of TTE in the crash reproduction experiment. Table 2 shows the results. We mark the result as N.A. if the fuzzer was not able to reproduce the target crash within 24 hours for more than half of the trials (out of 40 iterations) as the median is not available.

Overall, DAFL shows the best performance in 27 cases, while AFL, AFLGo, and WindRanger do so for 6, 0, and 4 cases, respectively. Also, DAFL was able to report a median TTE (i.e., did not result in N.A.) on 31 benchmarks, whereas AFL, AFLGo, and WindRanger reported a median TTE on 28, 25, and 21 benchmarks respectively. For 20 benchmarks where all the tools successfully reported a median TTE, DAFL outperformed each baseline by 1.93, 4.99, and 20.08 times, on average, respectively. Furthermore, DAFL signifi-
Table 1: Benchmark composition. The **Type** column indicates the type of vulnerability of each CVE with the following abbreviations: BOF (Buffer Overflow), ND (Null Dereference), SO (Stack Overflow), UAF (Use-After-Free), IO (Integer Overflow). The **Total #** row denotes the total number of programs for the **Program** column, the total number of CVEs for the **CVE** column, and the total number of the types of vulnerabilities for the **Type** column.

<table>
<thead>
<tr>
<th>Project</th>
<th>Program</th>
<th>Version</th>
<th>CVE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ming</td>
<td>swftophp</td>
<td>0.4.7</td>
<td>2016-9827</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-9829</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-9831</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.4.8</td>
<td>2017-9988</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-11728</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-11729</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-7578</td>
<td>BOF</td>
</tr>
<tr>
<td>Lrzip</td>
<td>lrzip</td>
<td>0.631</td>
<td>2018-8846</td>
<td>UAF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2018-11496</td>
<td>UAF</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>2.6</td>
<td></td>
<td>2016-4487</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-4489</td>
<td>UAF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-4490</td>
<td>IO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-4491</td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-4492</td>
<td>IO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2016-6131</td>
<td>SO</td>
</tr>
<tr>
<td>Bintuils</td>
<td>objcory</td>
<td>2.8</td>
<td>2017-8393</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-8394</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-8395</td>
<td>ND</td>
</tr>
<tr>
<td>objdump</td>
<td>2.8</td>
<td></td>
<td>2017-8392</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-8396</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-8397</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-8398</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td>2.31.1</td>
<td></td>
<td>2018-17360</td>
<td>BOF</td>
</tr>
<tr>
<td>strip</td>
<td>2.7</td>
<td></td>
<td>2017-7303</td>
<td>ND</td>
</tr>
<tr>
<td>nm</td>
<td>2.9</td>
<td></td>
<td>2017-14940</td>
<td>OOM</td>
</tr>
<tr>
<td>readelf</td>
<td>2.9</td>
<td></td>
<td>2017-16828</td>
<td>IO</td>
</tr>
<tr>
<td>Libxml2</td>
<td>xmlint</td>
<td>2.9.4</td>
<td>2017-5969</td>
<td>ND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-9047</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2017-9048</td>
<td>BOF</td>
</tr>
<tr>
<td>Libjpeg</td>
<td>cjpeg</td>
<td>1.5.90</td>
<td>2018-14498</td>
<td>BOF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2020-13790</td>
<td>BOF</td>
</tr>
<tr>
<td>Total #</td>
<td></td>
<td></td>
<td>10</td>
<td>41</td>
</tr>
</tbody>
</table>


In the supplementary experiment without ASAN, DAFL outperformed Beacon in 32 cases. In terms of median TTEs, DAFL was able to report a median TTE for 23 benchmarks, whereas Beacon was able to do so in 18 benchmarks. For the cases where DAFL and Beacon could both report median TTEs, DAFL was 20.34 times faster than Beacon on average.

DAFL consistently outperforms the other tools even when comparing only the fuzzing time, without considering the static analysis overhead. In terms of fuzzing time alone, DAFL showed the best performance for 30 cases in the main experiment and 32 cases in the supplementary experiment. For 20 benchmarks where all the tools successfully reported a median TTE, DAFL was 2.80, 2.69, and 22.01 times faster than AFL, AFLGo, and WindRanger, respectively, on average. Compared to Beacon, DAFL was 4.31 times faster on average for the benchmarks where they both successfully reported a median TTE.

The results also show that our static analysis is more cost-effective than the other approaches. Especially, Beacon spent a huge amount of time running the static analysis on cxxfilt. We conjecture that this is mainly due to the difficulty of the weakest precondition analysis. On the other hand, DAFL relies on a DUG which is easier to obtain. As a result, our analysis is 32.28 times faster than AFLGo, and 44.05 times faster than Beacon, on average.

Although DAFL is effective in most cases, there are 4 non-negligible cases where we underperformed: CVE-2018-20427, CVE-2019-9114, CVE-2017-8396, and CVE-2017-14940. This result is mainly due to the fact that our slicing does not soundly compute the control dependencies, as we discussed in §4.1. While our design of selective instrumentation successfully identified *intra-procedural* control dependencies in most cases, it did not fully capture *inter-procedural* control dependencies.

The case study of CVE-2018-20427 explains our limitation in detail. Our slicing result did not include `decompileAction` function, which handles various switch cases as shown in Figure 6 in the appendix. To trigger the target bug in Line 29, `push` and `decompileGETPROPERTY` functions have to be sequentially called from `decompileAction`. Thus, including `decompileAction` in the instrumentation target is important for discovering an input that explores these functions. However, `decompileAction` is related to the target location via inter-procedural control dependencies, not data dependencies. While `decompileAction` calls `decompileGETPROPERTY` with two arguments, `decompileGETPROPERTY` uses neither of these arguments when calling `getInt`. Instead, it pops idx from a global array and uses it as the argument of `getInt`. Thus, there is no data dependency between `decompileAction` and `decompileGETPROPERTY`. Other
Table 2: Crash reproduction results of DAFL and the baseline tools. $T_f$ denotes the time purely spent in the fuzzing process. $T_{f+sa}$ is the total time spent, considering both static analysis and fuzzing. Note that we only report $T_f$ for WindRanger as its static analysis time is negligible. Each reported number is a median over 40 repeated experiments. N.A. indicates that the tool could not produce a median TTE, which means that it was not able to reproduce the bug for more than half of the repeated experiments. The parentheses denote how many times the fuzzer was able to reproduce the bug. Unlike TTE, this number is better when bigger. For each target (i.e., each row), the best result is marked with bold font and an asterisk. # Best perf. denotes the number of targets for which the tool has the best performance among all the other tools.

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE</th>
<th>With ASAN</th>
<th>Without ASAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AFL</td>
<td>AFLGo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$T_f$</td>
<td>$T_f$</td>
</tr>
<tr>
<td>swftophp</td>
<td>N.A. (0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swftophp</td>
<td>N.A. (0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swftophp</td>
<td>N.A. (0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swftophp</td>
<td>N.A. (0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swftophp</td>
<td>3,362</td>
<td>2,662</td>
<td>3,009</td>
</tr>
<tr>
<td>swftophp</td>
<td>40,725</td>
<td>-</td>
<td>N.A. (13)</td>
</tr>
<tr>
<td>swftophp</td>
<td>1,463</td>
<td>3,118</td>
<td>3,326</td>
</tr>
<tr>
<td>swftophp</td>
<td>62,171</td>
<td>-</td>
<td>N.A. (11)</td>
</tr>
<tr>
<td>lrzip</td>
<td>N.A. (0)</td>
<td>-</td>
<td>N.A. (0)</td>
</tr>
<tr>
<td>lrzip</td>
<td>11</td>
<td>12</td>
<td>185</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>560</td>
<td>571</td>
<td>776</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>1,105</td>
<td>1,181</td>
<td>1,384</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>269</td>
<td>255</td>
<td>460</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>1,463</td>
<td>3,118</td>
<td>3,326</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>613</td>
<td>-</td>
<td>N.A. (0)</td>
</tr>
<tr>
<td>objcopy</td>
<td>1,425</td>
<td>1,399</td>
<td>3,054</td>
</tr>
<tr>
<td>objcopy</td>
<td>826</td>
<td>878</td>
<td>2,525</td>
</tr>
<tr>
<td>objcopy</td>
<td>8395</td>
<td>* 115</td>
<td>105</td>
</tr>
<tr>
<td>objdump</td>
<td>234,835</td>
<td>28,607</td>
<td>30,403</td>
</tr>
<tr>
<td>objdump</td>
<td>1,857</td>
<td>2,620</td>
<td>4,426</td>
</tr>
<tr>
<td>objdump</td>
<td>14,736</td>
<td>N.A. (2)</td>
<td>- N.A. (0)</td>
</tr>
<tr>
<td>strip</td>
<td>1,22</td>
<td>125</td>
<td>3,691</td>
</tr>
<tr>
<td>nm</td>
<td>19,582</td>
<td>30,009</td>
<td>33,443</td>
</tr>
<tr>
<td>readeif</td>
<td>221</td>
<td>192</td>
<td>729</td>
</tr>
<tr>
<td>readeif</td>
<td>5,969</td>
<td>* 320</td>
<td>395</td>
</tr>
<tr>
<td>xmlint</td>
<td>9,047</td>
<td>N.A.(19)</td>
<td>- N.A.(14)</td>
</tr>
<tr>
<td>xmlint</td>
<td>9,048</td>
<td>N.A. (2)</td>
<td>- N.A. (7)</td>
</tr>
<tr>
<td>cjpeg</td>
<td>14,498</td>
<td>N.A.(16)</td>
<td>- N.A.(12)</td>
</tr>
<tr>
<td>cjpeg</td>
<td>13,790</td>
<td>N.A. (6)</td>
<td>- N.A. (3)</td>
</tr>
<tr>
<td># Best perf.</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
cases where DAFL underperformed also have similar reasons. For example, our slicing missed `dump_dwarf_section` for CVE-2017-8396, which is related to the target location via inter-procedural control dependency. In short, the limitation of our slicing is that it sometimes fails to capture inter-procedural control dependencies. As a result, DAFL may not be able to recognize program behaviors that are crucial to trigger the target bug.

Despite the current limitation, we can improve our system by adopting better slicing techniques. To confirm this idea, we manually identified the missed functions that play a crucial role in the 4 CVEs where DAFL underperformed. Then, we added these functions as our selective instrumentation target. Consequently, we observed significant performance improvements of DAFL in CVE-2018-20427, CVE-2019-9114, and CVE-2017-14940 with the median TTE of 3802s, 16358s, and 1812s, respectively. Furthermore, DAFL showed comparable performance to the other tools for the remaining CVE: it resulted in N.A.(2) for CVE-2017-8396. This result confirms our hypothesis: better slicing will improve the performance of DAFL.

In summary, the experimental results demonstrate that our techniques can significantly improve the performance of directed fuzzing. Note that we could not perform the Mann-Whitney U test on the measured TTEs, since the fuzzer often resulted in a timeout during the repeated experiments. Such a timeout only tells that its TTE is longer than 24 hours, so this result cannot be used as a sample for the Mann-Whitney U test. Still, we believe that a median TTE computed from 40 repetitions already mitigates the randomness of fuzzing significantly. We also observed several limitations in our static analysis and slicing, but they can be orthogonally addressed by techniques actively being developed in the static analysis community. Thus, we leave it as future work to further improve our static slicing part.

**RQ1:** DAFL shows the best performance for 27 out of 41 cases, and reproduces crashes at least 1.93 times faster on average than all the baseline fuzzers.

### 5.3 Impact of Thin Slicing
Recall from §4.1 that the thin slicing approach returns a smaller set of dependent nodes than a naive one. To investigate the impact of thin slicing, we additionally instantiated DAFL with the naive slicing strategy, denoted as DAFL\textsubscript{Naive}. We compared the performance of DAFL, DAFL\textsubscript{Naive}, and AFL on 31 bugs in our benchmark where DAFL was able to reproduce them in more than half of the repeated experiments.

Figure 7 shows the comparison results. Overall, both DAFL and DAFL\textsubscript{Naive} outperformed AFL in most cases. This is because both slicing strategies can effectively filter out irrelevant functions. Table 3 in the appendix shows that even the naive slicing can reduce the number of instrumented functions by 57% on average while the thin slicing provides more effective filtering by reducing the functions by 76% on average.

When comparing DAFL and DAFL\textsubscript{Naive}, we observed that DAFL outperforms DAFL\textsubscript{Naive} in most cases, and DAFL was always more efficient than DAFL\textsubscript{Naive} in terms of

<table>
<thead>
<tr>
<th>Program</th>
<th># CVE</th>
<th>All</th>
<th>Naive</th>
<th>Thin</th>
</tr>
</thead>
<tbody>
<tr>
<td>swftophp</td>
<td>17</td>
<td>786</td>
<td>429</td>
<td>234</td>
</tr>
<tr>
<td>lrzip</td>
<td>2</td>
<td>154</td>
<td>112</td>
<td>77</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>6</td>
<td>211</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>objcopy</td>
<td>3</td>
<td>1478</td>
<td>1184</td>
<td>891</td>
</tr>
<tr>
<td>objdump</td>
<td>5</td>
<td>1787</td>
<td>1347</td>
<td>875</td>
</tr>
<tr>
<td>strip</td>
<td>1</td>
<td>1485</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>nm</td>
<td>1</td>
<td>1306</td>
<td>1017</td>
<td>50</td>
</tr>
<tr>
<td>readelf</td>
<td>1</td>
<td>441</td>
<td>185</td>
<td>112</td>
</tr>
<tr>
<td>xmllint</td>
<td>3</td>
<td>2414</td>
<td>1215</td>
<td>416</td>
</tr>
<tr>
<td>cjpeg</td>
<td>2</td>
<td>101</td>
<td>52</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 3: Slicing results for our benchmark. The **Program** column indicates the name of the program. The # **CVE** column is the number of CVEs in the program. The **All** column represents the number of all reachable functions in the program. The **Naive** and **Thin** column is for the average number of relevant functions sliced by the naive slicing and the thin slicing, respectively.
static analysis overhead: 2.04 times faster on average. The static analysis overhead was particularly significant in CVE-2017-8298: DAFL outperformed DAFL\textsubscript{Naive} even though DAFL\textsubscript{Naive} could be faster than DAFL in terms of fuzzing time. In total, DAFL reproduced the target crashes 2.01 times faster than DAFL\textsubscript{Naive} on average. Furthermore, the \textit{NA} cases where AFL failed to report median TTEs, were completely resolved by DAFL, unlike DAFL\textsubscript{Naive}. The overall results demonstrate that thin slicing provides more effective guidance for directed fuzzing.

RQ2: Thin slicing improves the fuzzing performance by 2.01 times on average compared to the naive approach.

5.4 Impact of Selective Coverage Instrumentation & Semantic Relevance Scoring

In this section, we investigate how each of our two main ideas affects the overall performance of directed fuzzing. To this end, we instantiate the following two variants of DAFL: 1) DAFL\textsubscript{SelInst}, which is the same as AFL but only employs selective coverage instrumentation; and 2) DAFL\textsubscript{SemRel}, which is the same as AFL but only employs semantic relevance scoring. We compared the performance of DAFL to these variants as well as the vanilla AFL on the same 31 bugs used in §5.3.

Figure 8 shows the results. Overall, each technique alone clearly gives positive guidance to AFL, and the combined use of them gives better results than either of them alone. DAFL\textsubscript{SelInst} and DAFL\textsubscript{SemRel} are respectively 1.73 times and 1.39 times faster than AFL. Note that even in the case where one of our techniques cannot outperform AFL (e.g., DAFL\textsubscript{SemRel} for CVE-2018-11225), using both techniques together help DAFL outperform AFL. This result demonstrates the synergetic effect of those two techniques. When combined, both techniques help DAFL to reproduce the bugs 1.93 times faster than AFL on average. Furthermore, significant improvements were made in 4 cases where AFL resulted in a timeout for more than half of the repetitions. DAFL resolved all 4 of these cases, while DAFL\textsubscript{SelInst} and DAFL\textsubscript{SemRel} could not when used individually.

We further investigated the impact of the subcomponents of Semantic Relevance Scoring: seed pool management and energy assignment scheme. We instantiate the variants of DAFL as follows: 1) DAFL\textsubscript{SeedPool}, which is the same as AFL but only employs our energy assignment scheme; and 2) DAFL\textsubscript{Energy}, which is the same as AFL but only employs our energy assignment scheme.

Figure 9 shows the results. Overall, seed pool management and energy scheduling both outperform AFL, each by 1.47 and 1.08 times on average, respectively. This result demonstrates that both seed pool management and energy assignment scheme are effective in improving the performance of directed fuzzing. DAFL\textsubscript{SeedPool} and DAFL\textsubscript{Energy} not only outperforms AFL separately, but is also synergistically combined into DAFL\textsubscript{SemRel}. While DAFL\textsubscript{SeedPool} and DAFL\textsubscript{Energy} result in timeout for 3 and 4 bugs, respectively, the combined version, DAFL\textsubscript{SemRel} results in timeout for only 1 bug.

Finally, we compared the effectiveness of the energy assignment scheme of DAFL\textsubscript{Energy} to AFLGo. Note that DAFL\textsubscript{Energy} differs from AFLGo in that it uses data dependency to assign energies for seeds, while AFLGo uses CFG-based distance. When compared to AFLGo, DAFL\textsubscript{Energy} reproduced the target crashes 1.27 times faster on average. Note that this comparison does not involve static analysis time in order to directly compare the effectiveness of energy scheduling. The comparison result demonstrates that energy scheduling based on data dependency is more effective than the one based on CFG distance.
RQ3: Selective Coverage Instrumentation and Semantic Relevance Scoring each improves the performance by 1.73 and 1.39 times. When combined, DAFL shows 1.93 times faster crash reproduction results compared to AFL.

6 Discussion

Supporting Multiple Targets In the scope of this paper, we focused on single-target directed fuzzing as in prior work [23, 32]. However, DAFL can be naturally extended to support multiple targets. The underlying static analysis will be the same as before; it will be performed only once for a given program. Instead, the slicing part will be separately performed for each target. Then, we can selectively instrument functions in the union of the slices and generate a single binary program. Also, the semantic relevance score would be calculated by aggregating the scores from the individual slices (e.g., taking the average). We leave this extension as future work.

Supporting Other Languages DAFL currently supports C programs, but the underlying principle can be extended to support other languages. The effectiveness of thin slicing [57] was originally demonstrated with Java and there is a large body of work that apply program slicing to object-oriented programs [2, 6, 33, 48]. By adopting such techniques, we believe that DAFL can be extended to support object-oriented languages. However, there may exist challenges in other languages with more complex features such as C++. The dynamic nature of C++ often causes the pointer analysis to be imprecise in practice [37, 38]. This imprecision can lead the slicing results to either lack important functions or include unnecessary functions. We leave these challenges for supporting other languages as future work.

7 Threats to Validity

Although we carefully designed the evaluation criteria such as our bug triage logic (§5.1), they may not be exactly the same as the existing work. As mentioned in §1, it is exceedingly
difficult to replicate their settings in the absence of detailed information. In order to mitigate the threat, we have communicated with the authors and carefully configured the settings to reproduce the partial results from the prior work.

The performance of fuzzers may vary due to the experimental environment, such as the versions of the baselines and underlying compiler toolchains. Since different tools are based on different versions of software, we commonly set up the main components with the latest versions. This change may introduce discrepancies in the results. For example, one might wonder why Beacon is less effective than AFLGo in most cases, contrary to the reported performance in the paper. One reason is that we used the latest version of AFLGo which is more effective than the old version used in the Beacon paper. We also used LLVM 12 while the Beacon paper used LLVM 4. This might have an impact on the performance of AFLGo because AFLGo relies on the control flow graph generated by the LLVM compiler.

Due to the inherent randomness posed by fuzzing, our experimental results can have significant fluctuations. To reduce the effect, we repeated all our experiments 40 times and reported the median values.

8 Related Work

Fuzzing, especially Grey-box fuzzing [8, 24, 41] has become a critical method to discover bugs in real-world software such as parsers [44], web browsers [28, 61], network protocols [26, 36], autonomous driving cars [35], mobile apps [39], and OS kernels [19, 34, 52]. However, generating a test case for a particular bug in a program remains challenging. Directed Grey-box Fuzzing (DGF) tackles the challenge by favoring test cases that can exercise nodes closer to the target node in the Control-Flow Graph (CFG). AFLGo [9] is one of the earliest directed fuzzers leveraging this idea. Similarly, Hawkeye [15] and WindRanger [23] improve upon AFLGo to provide more informative feedback to precisely compute distances. ParmeSan [47] considers sanitizer-guided bug coverage in addition to the distance feedback. FuzzGuard [64] also enhances AFLGo by leveraging deep learning to filter out irrelevant test cases. Likewise, TargetFuzz [13] comprises a target-oriented seed corpus to enhance the effectiveness of AFLGo. MC² [55] takes a unique approach that leverages binary search rather than mutation to generate more promising inputs toward the target. Note that all these fuzzers rely on a distance-based mechanism, which suffers from the precision problem, and DAFL is the first in tackling it.

There exist DGF approaches that aim to improve efficiency by transforming the program itself. Beacon [32] and SieveFuzz [58] aggressively prune off irrelevant paths on runtime by modifying the program under test. However, DAFL is orthogonal to these approaches, as it can distinguish not only irrelevant and relevant paths but less relevant and more relevant paths as well.

Directed White-box Fuzzing, or Directed Symbolic Execution (DSE) [40], is a predecessor of DGF, which views the target-reaching problem as a constraint satisfaction problem. It has been used for various purposes such as patch testing [43, 49] and verification of static analysis reports [21]. More recently, directed hybrid testing was proposed to combine symbolic execution and fuzzing [17]. However, these approaches rely on symbolic execution [11, 27, 53], which is orthogonal to our approach.

There have been numerous data-flow-guided fuzzers [14, 16, 18, 25, 47, 50]. However, semantic relevance scoring is unique in that it uses a data-flow analysis to compute relative distances between program executions, which will then be used for driving fuzzers toward buggy lines of code. Notably, GREYONE [25] employs dynamic taint analysis to guide the mutation process and prioritize seeds. However, its dynamic analysis deals with a single execution path at a time, whereas our static analysis examines the whole program without concrete executions. This allows DAFL to guide fuzzing toward an unreached target location.

Static analyses often help improve the performance of fuzzers. DGF calculates seed distances by statically analyzing the control-flow of the target program [9]. Shastry et al. [56] automatically generate a dictionary of inputs for a target program based on static analysis. NTFuzz [19] leverages a static type inference technique to guide kernel fuzzing. VeriFuzz [20] uses static analysis to transform the loops in the program to be fuzzed more easily. Furthermore, it determines the range of valid inputs with interval analysis. DDFuzz [42] and DATAFLOW [30] introduces a new feedback system, by receiving feedback not only from discovering a new control-flow edge, but also from discovering a new data-flow edge. They perform intra-procedural program analysis to obtain such data-flow graph. Unlike these works, we selected only the data-flow edges that are relevant to the target bug. For this, DAFL must perform inter-procedural data-flow analysis to collect all the relevant def-use chains across the function boundaries. Moreover, we further process the obtained def-use chains to compute the semantic relevance scores of seeds.

Our approach uses thin slicing [57] to identify relevant parts of the program to the target location. Unlike traditional static program slicing [60, 62], thin slicing only includes producer statements that define non-pointer values, in our context. Since fuzzers typically attempt to mutate non-pointer values, thin slicing results provide concise yet useful guidance to the target locations. We demonstrated that thin slicing is more effective than traditional approaches for directed fuzzing.

9 Conclusion

We presented DAFL, a new directed grey-box fuzzer. We addressed two technical challenges in DGF. First, code coverage can give negative feedback to fuzzers when they exercise irrelevant paths to the target point. Second, syntactic distance met-
rics can mislead fuzzers when complicated control flows, such as loops, are involved. Our idea is to selectively receive coverage feedback only from the relevant parts to the target buggy location and prioritize seeds based on the semantic relevance score defined on the Def-Use Graph. Our empirical results show that DAFL significantly outperforms existing state-of-the-art directed fuzzers, including AFLGo, WindRanger, and Beacon.

Acknowledgements

We thank Soomin Kim for his sincere advice in conducting the experiments. This work was partly supported by (1) the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1C1C1003876, NRF-2021R1A5A1021944, and RS-2023-00210989); and (2) Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains).

References


