On the Effectiveness of Synthetic Benchmarks for Evaluating Directed Grey-box Fuzzers

Haeun Lee, Hee Dong Yang, Su Geun Ji, and Sang Kil Cha
Cyber Security Research Center
KAIST
Daejeon, Korea
{haeunlee, heedong, jsgtwins1, sangkilc}@softsec.kaist.ac.kr

***Abstract*—Directed grey-box fuzzing is difficult to rigorously evaluate for several reasons. First, directed grey-box fuzzers are more prone to overfitting than undirected grey-box fuzzers as they are designed to explore specific paths in the program under test. Furthermore, existing benchmarks are mainly designed for evaluating undirected fuzzers. Hence, they do not provide any information about bug locations, and the difficulty of triggering bugs can substantially vary across different benchmarks. In this paper, we argue that one can address these challenges by automatically generating benchmarks with a bug synthesis technique. Notably, Fuzzle, a state-of-the-art bug synthesis tool, enables generation of arbitrarily many benchmarks, thereby preventing the overfitting problem. It is also well suited for evaluating directed grey-box fuzzers as it provides the exact location of the target bug in the generated benchmark with a guarantee that the bug is lurking deep in the program. With Fuzzle, we systematically evaluate existing state-of-the-art directed fuzzers and study their strengths and weaknesses, which would be otherwise difficult to obtain with traditional benchmarks. To our knowledge, this is the first attempt to adopt a bug synthesis technique for evaluating directed fuzzers.**

I. Introduction

Directed Grey-box Fuzzing (DGF) is gaining popularity in recent years due to its effectiveness in revealing bugs for targeted testing scenarios, such as crash reproduction and patch testing [3], [5], [11], [22], [33], [40]. At a high level, DGF gives higher priority to seeds that are likely to reach a target location in the program, such as a crash site or a patched location, in order to drive the fuzzer towards the target.

Despite its popularity, there has been little research on evaluating DGF solutions, which is a non-trivial task for several reasons.

First, DGF can easily overfit to the benchmarks used for evaluation because it is designed to explore specific paths in the program under test. Unlike undirected fuzzing, whose goal is in achieving the maximum code coverage, DGF has a specific target, making it trivial to derive heuristics that are specifically tailored to the target. That is, it is easy to come up with a DGF solution that performs well on a given set of benchmarks, but fails to generalize to others.

Second, existing benchmarks for evaluating fuzzers do not provide specific target locations to reach as they are designed for evaluating undirected fuzzers. For example, in order to evaluate a DGF tool on GNU Binutils or Google's FuzzBench [26], one needs to manually identify target bug locations or patched code locations from every program in the suite, which is a tedious and error-prone task.

Finally, existing benchmarks do *not* necessarily contain deep bugs that can only be reached by penetrating multiple layers of the program. For example, CVE-2016-9827 [29] used by recent directed fuzzers in their evaluation [17], [20] is a shallow bug that can be easily reached by traversing only three functions from the `main` function. In our preliminary study, we found that even the vanilla AFL can find the bug within a few minutes without any directed guidance. Therefore, simply using previously known bugs for evaluating DGF is not sufficient to test the effectiveness of DGF.

In this paper, we propose to use *synthetic benchmarks* to address the aforementioned challenges. While there has been significant research effort in synthesizing buggy programs for evaluating *undirected* fuzzers [10], [23], [31], [38], there has been no prior work for evaluating directed grey-box fuzzers. Thus, we hypothesize that current bug synthesis techniques can be used for evaluating DGF, and those evaluation results with synthetic benchmarks can provide useful insights for improving DGF, which would be otherwise difficult to obtain with traditional benchmarks.

To test our hypothesis, we use Fuzzle [23], one of the most recent bug synthesis tools, to generate synthetic benchmarks, and evaluate several state-of-the-art DGF tools on them. Fuzzle is suitable for our purpose for the following reasons. First, Fuzzle, by its design, is able to generate a large number of synthetic benchmarks that are independent to each other. This allows us to evaluate DGF tools on a new set of benchmarks whenever it is needed. Second, Fuzzle provides an exact location of the target bug in the generated benchmark with a guarantee that the bug can be triggered by a user input. Third, Fuzzle generates programs whose function call graph resembles a 2D maze, allowing us to visualize the current progress of the fuzzer in a more intuitive manner. With this, we can easily understand the behavior of fuzzers.

Additionally, the highly customizable nature of Fuzzle allows us to evaluate DGF tools in various settings. For example, one can vary the size of the benchmark program or the path constraints to reach the target bug to test the scalability of DGF tools. Nevertheless, we further modify Fuzzle to conduct more controlled experiments, e.g., creating benchmarks with a varying number of bugs per program.

Algorithm 1: Grey-box Fuzzing

```
Input  : Program $P$, Initial Seed Corpus $S$
Output: Augmented Seed Corpus $S$
1 while $\neg$TimeOut() do
2     $\forall s \in S$: AssignEnergy($s$)// based on coverage
3     $s \leftarrow$ SelectSeed($S$);
4     $s' \leftarrow$ Mutate ($s$);
5     $r \leftarrow P(s')$;
6     if $r$ increases coverage then
7         $S \leftarrow S \cup \{s'\}$;
8 return $S$;
```

With the modified Fuzzle we synthesize 12 new buggy programs that have various different program structures, sizes, and difficulties. We then *systematically* design a series of experiments with these benchmark programs to evaluate state-of-the-art directed grey-box fuzzers, including AFLGo [3], Beacon [17], and DAFL [20], and share several insights we have gained from the experiments.

Some of the key findings from our experiments are as follows. First, we find that synthetic benchmarks are indeed useful for evaluating DGF, and they can provide a clearer understanding than traditional benchmarks regarding the limitation of a given DGF tool. Second, we note that synthetic benchmarks enable us to identify the exact bottleneck of a DGF tool in an intuitive way. In particular, the visualization feature of Fuzzle allows us to identify the exact path conditions that prevent the fuzzer from reaching the target bug. Finally, we find that recent directed fuzzers are less affected by the size of the benchmark program than undirected fuzzers, which is indeed a desirable property for DGF.

In summary, our contributions are as follows:

1) We propose to use synthetic benchmarks for evaluating directed grey-box fuzzers, and show that it is a promising approach to address the challenges in evaluating DGF.
2) We modify Fuzzle, an existing bug synthesis tool, to generate benchmark programs for evaluating DGF.
3) We design a series of experiments with the synthesized benchmark to evaluate state-of-the-art directed grey-box fuzzers, and find novel insights that would have been difficult to obtain with traditional benchmarks.

II. Background

This section provides background information on directed grey-box fuzzing as well as bug synthesis techniques that are relevant to our work.

A. Directed Grey-box Fuzzing in a Nutshell

Grey-box fuzzing is an iterative process of generating and evaluating test cases while evolving the quality of the test cases over time based on a fitness criterion [25]. The most common criterion used in grey-box fuzzing is code coverage as shown in Algorithm 1. Typically, the initial seed corpus S is given by the user, and the corpus grows as the fuzzer generates new test cases that achieve more code coverage. The probability of selecting a seed s from the corpus (by `SelectSeed()`) is proportional to the energy assigned to s by the fitness criterion.

Directed Grey-box Fuzzing (DGF) is a variant of grey-box fuzzing that aims to find a test case reaching a specific target location in the program under test. Most existing DGF techniques modify `AssignEnergy()` to assign more energy to test cases that are likely to reach the target location. For example, AFLGo [3] computes the average distance between the currently executed nodes and the target node in the Control-Flow Graph (CFG) to obtain the fitness score of a test case. The follow-up studies, such as Beacon [17], WindRanger [11], and DAFL [20], share a similar design philosophy. They employ various different fitness functions to give more effective guidance to grey-box fuzzing. To distinguish DGF from other grey-box fuzzing techniques, we call the latter as *undirected* grey-box fuzzing (or undirected fuzzing) throughout this paper.

B. Bug Synthesis and Fuzzle

Overfitting has been a long-standing problem in various fields of computer science [12], and fuzzing is no exception. In the context of fuzzing, overfitting refers to the phenomenon that a fuzzer performs well on a specific set of benchmarks but poorly on other benchmarks. For example, most fuzzing papers today evaluate their techniques on a set of well-known benchmarks, such as Fuzzer-Test-Suite [18] and GNU Binutils, but it is unclear whether the techniques can generalize to other benchmarks.

Bug synthesis is a recently emerging technique that can be used to address the overfitting problem in fuzzing by automatically generating benchmarks on demand. The idea is to always use newly generated benchmarks for evaluating a fuzzer so that the fuzzer cannot overfit to a specific one. LAVA [10] is one of the first bug synthesis tools, which injects bugs into an existing program to generate new benchmarks. The follow-up works, such as Apocalypse [31], FixReverter [38], and EvilCoder [30], follow the same idea of injecting arbitrary bugs into existing programs.

While injection-based bug synthesis is a promising approach, it suffers from several limitations. First, it is challenging to know whether the injected bugs are reproducible. Second, it is difficult to know whether the original program is free of bugs. Hence, there could be existing bugs in the original program, which can lead to false alarms in the evaluation of a fuzzer. Third, it is unclear whether the injected bugs will affect the behavior of the original program in such a way that unintended bugs are introduced.

To address the limitations of injection-based bug synthesis, Fuzzle [23] proposes a way to synthesize a whole program from scratch. The core idea is to cast the bug synthesis problem as a maze generation problem, where finding a path from the entry to the exit in the maze is transformed into finding a path to the bug in the program. More specifically, every cell in the maze corresponds to a function in the program, and every pathway between two cells corresponds to a guarded function call. Those guards are constructed by collecting path constraints from real-world programs using symbolic execution [32]. In this paper we leverage path constraints collected from real-world CVEs as in [23].

The key advantage of Fuzzle is that a synthesized program is guaranteed to have reproducible bug(s). Furthermore, the location of the injected bug is explicit as it is represented as a function in the maze, and we can see the current progress of the fuzzer by visualizing the maze. Such characteristics of Fuzzle make it a promising tool for evaluating directed grey-box fuzzers.

III. Experiment Design

This section describes our experimental design. We first present how we modify Fuzzle and configure it to generate synthetic benchmarks for evaluating directed grey-box fuzzers. We then describe the research questions we aim to answer with our experiments.

A. Extending Fuzzle

While Fuzzle provides a convenient way to generate synthetic benchmarks, the current implementation of Fuzzle does not allow us to control the number of bugs in the generated program, which are crucial factors in evaluating directed grey-box fuzzers as we will discuss in §III-B. To enable such additional configurability, we made several modifications on Fuzzle, which comprises of 144 lines of Python code.

Specifically, we have modified Fuzzle to insert variable number of buggy functions, `func_bug` which calls the `abort` system call, in addition to the default `func_bug` that is placed at the exit of the maze. Note that we cannot randomly place the buggy functions in the program because doing so can cause the program to terminate unexpectedly and block off one or more areas in the program for further exploration. This may even cause other bugs in the program to be unreachable which becomes problematic when evaluating fuzzers. Randomly placing the buggy functions can also lead to other problems, such as having some bugs be much easier to find than others and having all bugs in close proximity to each other, which can impede the thorough evaluation of directed fuzzing with multiple targets.

To avoid the aforementioned problems, we set the following rules when selecting the candidate positions for additional buggy functions. First, we place the additional `func_bug` only at one of the dead-ends of the maze. This ensures that the newly added buggy function does not cut off any path in the generated maze. From the available dead-ends in the maze, we then select only those that are of the same distance from the entrance as the original default `func_bug`. In other words, we only use dead-ends that are *solution-length* away from the entrance of the maze, where a *solution length* is the length of the path from the entry of the maze to the exit of the maze. By doing so, we generate multiple bugs of similar difficulties. Lastly, from the dead-ends that satisfy the above criteria, we select the farthest one from the initial `func_bug`. We additionally check that all bugs in the program are at least maze's width away from each other to reduce the impact of directing the fuzzer towards one bug on finding the other bugs. Note that we measure the distance between two dead-ends by measuring the length of the path between them. For our evaluation of multi-target directed fuzzing, we generated two programs that contain three bugs each, including the default bug at the exit of the maze.

B. Research Questions

We aim to answer a series of research questions with our experiments. We start by asking whether synthetic benchmarks are suitable for evaluating directed grey-box fuzzers.

RQ1. Does evaluating DGF on synthetic benchmarks demonstrate their capability in directing fuzzers? (§IV-B)

To quantitatively measure the effectiveness of synthetic benchmarks on DGF, we run both directed and undirected fuzzers on the synthetic benchmark we generated and compare their performance. This will additionally help us understand the effectiveness of existing DGF techniques.

RQ2. How do the synthetic programs compare to the real-world programs in evaluating DGF? (§IV-C)

The follow-up question to RQ1 is whether synthetic benchmarks are comparable to organic benchmarks consisting of real-world programs in terms of evaluating directed fuzzers. While existing work on directed fuzzing has only used organic benchmarks to evaluate their techniques, they often measure the performance improvement that their techniques bring over the undirected baseline fuzzer. Therefore, we compare the relative performance improvement of using DGF techniques on both synthetic and organic benchmarks to understand the value of using synthetic benchmarks.

RQ3. How does the size of benchmark programs affect the effectiveness of directed fuzzers? (§IV-D)

As DGF guides the fuzzing process towards exploring a specific target location, we conjecture that the size of the program should have little to no impact on the performance of directed fuzzers. More specifically, we hypothesize that the impact of the program size on the performance of directed fuzzers is less than that on the performance of traditional undirected fuzzers, whose goal is to achieve maximum code coverage. We answer this research question by synthesizing four differently sized programs with Fuzzle, and by comparing the performance of directed fuzzers on these four programs. Note that merely selecting four random real-world programs of different sizes is not sufficient to answer this research question because it is extremely difficult to make those programs to have consistent behaviors and similar difficulties in terms of finding bugs.

RQ4. How do the path conditions towards the target location affect the effectiveness of directed fuzzers? (§IV-E)

A natural question that follows from RQ3 is whether the difficulty of the path conditions towards the target location has any impact on the performance of DGF. In contrast to the impact of the program size, the path conditions to satisfy to reach the target location could have significant impact on the performance of grey-box fuzzers. Although DGF techniques drive the fuzzer towards the target location, they do not provide any assistance in generating inputs that satisfy difficult constraints. Based on this, we hypothesize that varying the path constraints along the path to the target location will have similar impact on both the directed and undirected fuzzers. To answer this research question, we use seven different programs synthesized with Fuzzle, each with different path constraints towards the target bug. Six of them leverage the path constraints from previous CVEs, and the remaining one employs simple one-byte range checks. We make sure to keep all other parameters of Fuzzle same across all programs, which allows us to study the sole impact of path constraints on the performance of directed fuzzers.

RQ5. What is the impact of having multiple target locations on the effectiveness of directed fuzzers? (§IV-F)

As some directed fuzzers, including AFLGo and Hawkeye [6], support targeting multiple locations in the program, we want to study the effectiveness of directed fuzzers when provided with more than one target location. Specifically, we check whether running directed fuzzers with multiple targets is more effective than running multiple instances of directed fuzzers, each with a single target. To answer this research question, we generate programs with three bugs using extended Fuzzle (as described in §III-A) while keeping their depths the same. This ensures that all three bugs are equally difficult to reach, which is infeasible to achieve with real-world benchmarks.

RQ6. Can the visualization feature of Fuzzle help in understanding the bottleneck of directed fuzzers? (§IV-G)

One of the useful features of Fuzzle is that it provides coverage visualization of the fuzzing results. Lee *et al.* [23] have demonstrated the usefulness of visualized coverage in terms of showing the progress of different fuzzers as well as the fuzzing progress over time. In this paper, we want to study whether the visualization feature of Fuzzle can also be useful in understanding the performance bottleneck of directed fuzzers, which can potentially provide new insights into improving the performance of DGF techniques.

IV. Evaluation and Lessons Learned

In this section, we aim to answer the research questions presented in III-B and share several insights we have gained from the experiments.

A. Experimental Setup

1) Fuzzers: For our experiment, we selected a total of five fuzzers including both traditional coverage-guided (undirected) grey-box fuzzers and directed grey-box fuzzers. Specifically, we included AFL [37] (v2.57b) and AFL++ [13] (v4.07c) as they are the most widely used fuzzers in research and industry. As for the directed fuzzers, we used three state-of-the-art tools: AFLGo [3] (ac9246a), Beacon [17] (v1.0.0), and DAFL [20] (a6fcc56). Note we use the most up-to-date versions of the tools at the time of writing.

TABLE I: Numbers of benchmark programs generated with Fuzzle for each different configuration. We generated a total of 12 benchmark programs with Fuzzle.

	Single Bug				**Three Bugs**	
Constraints	**10×10**	**20×20**	**30×30**	**40×40**	**20×20**	**30×30**
CVE-2016-4487		1				
CVE-2016-4489		1				
CVE-2016-4491		1				
CVE-2016-4492		1				
CVE-2016-4493		1				
CVE-2016-6131		1				
Range checks	1	1	1	1	1	1

For a fair comparison, we ran all the fuzzers by instrumenting the source code of our benchmark programs as all the directed fuzzers we used require the source code to operate. We also used the same initial seed corpus with a single empty seed for all fuzzing campaigns.

2) Fuzzer Configuration: We fine-tuned the configuration of each fuzzer for fair comparison. In particular, we turned off the deterministic mode of AFL and AFLGo (with `-d`) because all the other fuzzers explicitly turn it off in their default configuration. For AFLGo, we set the time-to-exploitation to 18 hours in accordance with the authors' recommendation of 3/4 of the total run time. For both AFL++ and Beacon, we used the default configuration provided by the authors. For DAFL, we set `-max_pre_iter` to 100 (instead of 10) in order to successfully perform the preprocessing step on our benchmark programs.

3) Benchmark: We used the modified version of Fuzzle (as described in §III-A) to construct our benchmark with a variety of configurations described in Table I. Specifically, we have generated 12 benchmark programs with seven different constraints, 4 different program sizes (10x10, 20x20, 30x30, and 40x40), and two different numbers of synthetic bugs (one bug or three bugs per program). We used a smaller maze size of 20x20 as the default size instead of 30x30 from the original paper of Fuzzle to enable better comparison between the fuzzers as AFLGo and AFL struggled to find any bugs in most of the programs of size 30x30; Lee *et al.* [23] made the same observation in their paper. We used the default options for the rest of the parameters. Note that we used the same CVEs from the original paper of Fuzzle [23]. As Fuzzle generates programs with a precise ground truth in terms of the number and the location of bugs, we give the buggy line in the source code as the target for all the directed fuzzers.

4) Environments: We ran all the experiments on a server equipped with 88 Intel Xeon E5-2699 v4 CPU cores with 2.20 GHz and 128 GB of memory. Each fuzzing campaign was run in an isolated Docker container assigned with one CPU core and 8GB of memory. Every experiment was run for 24 hours and was repeated ten times.

TABLE II: The average Time-To-Exposure (TTE) in hours and the number of successful attempts out of ten repeated runs, for each fuzzer.

Measure	Fuzzer	CVE-2016-4487	CVE-2016-4489	CVE-2016-4491	CVE-2016-4492	CVE-2016-4493	CVE-2016-6131
TTE (h)	AFL	6.37	2.12	-	1.38	3.81	-
	AFL++	2.57	0.67	-	0.96	1.82	14.85
	AFLGo	4.23	2.23	-	1.15	3.25	22.64
	Beacon	3.31	0.96	-	0.87	1.05	12.87
	DAFL	1.27	0.61	-	0.50	1.28	16.19
# of hits over 10 runs	AFL	10	10	0	10	10	0
	AFL++	10	10	0	10	10	2
	AFLGo	10	10	0	10	10	2
	Beacon	10	10	0	10	10	2
	DAFL	10	10	0	10	10	7

B. RQ1: Usefulness of Synthetic Benchmarks in Evaluating Directed Fuzzers

How effective are synthetic benchmarks in evaluating directed fuzzers? To answer this question, we ran the three state-of-the-art directed fuzzers described in §IV-A1 along with two popular undirected fuzzers on the Fuzzle-generated programs. Table II summarizes the fuzzing results over six different benchmark programs with different path constraints obtained from six previous CVEs. The Time-To-Exposure (TTE) row represents the time taken to find the bug in each program. Each number is the arithmetic mean of 10 repeated runs. The lower part of the table shows the numbers of successful runs, where a fuzzer was able to hit the bug within a 24-hour fuzzing campaign. For example, every fuzzer was able to find the bug from CVE-2016-4489 in all 10 runs although the time taken to find the bug varied from minutes to hours.

Base on the results, we see that the directed fuzzers are indeed better at reaching the target locations when compared to the undirected fuzzers, and that there exist significant differences in performance even between the directed fuzzers. We highlight two interesting observations we made from the results as follows.

1) Impact of DGF techniques: The impact of DGF techniques is indeed higher than it of various optimizations adopted by AFL++ over the original AFL, such as sophisticated seed selection and mutation strategies [13]. This is clear because all the four fuzzers (AFL++, AFLGo, Beacon, and DAFL) are implemented on top of AFL. The average performance gains of AFL++, AFLGo, Beacon, and DAFL over AFL were 2.29×, 1.21×, 2.34×, and 3.56×, respectively. It is worth noting that the performance gain of AFLGo is much lower than that of the other two directed fuzzers. We believe this is due to the limitation of the seed distance metric used in AFLGo, as discussed in [17] and [20], which does not take into account the semantic relevance of the seed to the target location(s) in the program. The performance gains of

TABLE III: Comparing the performance gain reported in [20] and the results in Table II. Note that AFL is the baseline fuzzer.

Fuzzer	Bug Type	Source	#Progs.	Fuzz. Hours	Gain over AFL
AFLGo	Organic	Table 2 of [20]	25	24	×0.51
	Synthetic	Table II	4	24	×1.21
Beacon	Organic	Table 2 of [20]	17	24	×0.45
	Synthetic	Table II	4	24	×2.34
DAFL	Organic	Table 2 of [20]	27	24	×2.03
	Synthetic	Table II	4	24	×3.56

the more recent directed fuzzers, Beacon and DAFL, confirm that the directedness of the state-of-the-art directed fuzzers are indeed effective in terms of guiding the fuzzer towards the target location. Since the optimization techniques employed by AFL++ are orthogonal to the DGF techniques, we believe that combining the best of both worlds would further improve the state of the art in DGF.

Lesson 1. Synthetic benchmarks can clearly demonstrate the impact of DGF techniques over other general-purpose fuzzing techniques adopted by undirected fuzzers.

2) Comparing different directed fuzzers: The three directed fuzzers we used (AFLGo, Beacon, and DAFL) showed significant differences in their performance in reaching the target location, even though they all operate by giving direction to AFL. For example, Beacon and DAFL found most of the bugs within the first hour of fuzzing, whereas it took 2.72 hours on average for AFLGo. On average, Beacon and DAFL find bugs 2.00 times and 2.96 times faster than AFLGo, respectively. Therefore, the synthetic benchmarks generated by Fuzzle can be used to effectively evaluate and compare different directed fuzzers.

Lesson 2. Synthetic benchmarks can provide clear distinction between different directed fuzzers, which is useful in comparing their performances.

C. RQ2: Comparing Results on Synthetic Programs and Real-World Programs

To further understand the impact of synthetic benchmarks in evaluating DGF, we now compare the performance gains of DGF techniques on the synthetic benchmarks to those on the organic (i.e., real-world) benchmarks. Specifically, we compare the speed-ups observed in Table II to those reported in the recent study using organic benchmarks [20]. When calculating the speed-ups, we only included the benchmarks where both fuzzers found the bug in more than half of the experiment runs, following the approach used in [20]. We summarize the results in Table III. Note that this comparison is not meant to be a fair comparison as the experiments were conducted in different environments and with different setups. Rather, we aim to highlight the significance of using synthetic benchmarks in evaluating DGF techniques.

TABLE IV: Fuzzing results on Fuzzle-generated programs with varying sizes. The average time taken to find the bug (TTE) and the number of runs that found the bug within the 24 hours of fuzzing are reported for each fuzzer.

Measure	Fuzzer	10×10	20×20	30×30	40×40
TTE (h)	**AFL**	0.04	0.69	1.06	4.68
	AFL++	0.04	0.24	0.61	3.00
	AFLGo	0.03	0.64	2.07	4.66
	Beacon	0.02	0.21	0.24	1.82
	DAFL	0.05	0.18	0.31	0.97
# of hits over 10 runs	**AFL**	10	10	10	9
	AFL++	10	10	10	10
	AFLGo	10	10	10	10
	Beacon	10	10	10	10
	DAFL	10	10	10	10

Table III shows that the performance gain of the all three directed fuzzers in comparison to their baseline fuzzer (AFL) is higher when evaluated on the synthetic benchmarks than on the organic benchmarks. This means that the impact of DGF is more obvious with synthetic benchmarks than with organic benchmarks. Thus, the programs synthesized using Fuzzle are more suitable for evaluating the extent of directedness than the curated benchmarks commonly used in evaluating the undirected grey-box fuzzers.

Lesson 3. Synthetic benchmarks can give more clear picture of the performance gain of DGF techniques than the organic benchmarks.

D. RQ3: Impact of Program Size

Recall from §III-B, we hypothesized that the size of a program should not affect much the performance of directed fuzzers as the difficulty of reaching a specific target location should be independent of the size of the program, but rather dependent on the solution length, i.e., the distance between the entry point and the target location. To test this hypothesis, we used Fuzzle to generate programs with varying sizes (mazes of sizes 10x10, 20x20, 30x30,and 40x40) and ran both directed and undirected fuzzers on them. Note that for the rest of the evaluation, we used the equally-divided range checks for all branch conditions when generating the benchmarks, as it is the default configuration for Fuzzle. The results of running the fuzzers on the four generated programs are summarized in Table IV. The reported numbers for TTE are the arithmetic mean of 10 runs.

From the table, we see that the increase in the program size has a much greater impact on the performance of undirected fuzzers than directed fuzzers. For example, AFL and AFL++ took 109× and 83× longer, respectively, to find the bug in the program of the largest size (40x40) than in the program of the smallest size (10x10), whereas Beacon and DAFL took 76× and 21× times longer, respectively.

One notable exception is AFLGo, which took 173× longer to find the bug in the program of the largest size than in the program of the smallest size. AFLGo showed comparable or worse performance than that of AFL, which is the baseline undirected fuzzer, for the programs of the larger sizes (i.e., 30x30 and 40x40). This result indeed aligns with the results presented by the authors of Fuzzle in their evaluation of AFL and AFLGo on their benchmark [23]. Based on these results, we conjecture that the seed distance metric employed in AFLGo becomes less effective for guiding the fuzzer towards the target location in larger programs. Indeed, a similar observation has been made by the authors of DAFL [20] as well. Thus, we conclude that the size of a program does not affect much the performance of directed fuzzers as long as they can effectively direct the fuzzer towards the target location as in the case of *recent* directed fuzzers (Beacon and DAFL) we tested.

Lesson 4. Program size has greater impact on the performance of undirected fuzzers than recent directed fuzzers.

E. RQ4. Impact of Path Constraints

Recall from §III-B we hypothesized that varying path constraints should affect the performance of both directed and undirected fuzzers alike. As we have already ran the fuzzers on seven programs generated with different path constraints—six programs based on previous CVEs (§IV-C), and one program with one-byte range checks (§IV-D)—we can compare the results shown in Table II and Table IV to answer this research question. Note that the seven programs differ only in their path constraints along the buggy path as they were generated using the same maze.

By comparing the fuzzing results summarized in Table II and in the 20x20 column of Table IV, it is apparent that the complexity of path constraints in the program affect the performance of both directed and undirected fuzzers to a similar degree. First, all fuzzers, including the directed fuzzers, spent the least time to find the bugs in the programs with range checks. This is because the mutation-based fuzzers can easily generate inputs that satisfy simple range checks. On the other hand, many fuzzers did not perform well on the programs generated with more complex constraints from the CVEs (Table II). Particularly, all fuzzers failed to find the CVE-2016-4491 due to a large number of equality checks in the path towards the target bug. Indeed, the program generated using path constraints from CVE-2016-4491 had a total of 26 branches with one or more equality checks compared to the average of 7 branches for the rest of the programs in Table II. Thus, we see that the DGF techniques for guiding the fuzzer towards the target location do not help much the fuzzers in passing difficult path constraints. One notable exception is CVE-2016-6131, where DAFL showed significantly better performance compared to the others. We will discuss this case further in §IV-G.

Lesson 5. Path constraints of the program have comparable impact on the performance of both directed and undirected fuzzers.

TABLE V: Fuzzing results of AFLGo on programs with three bugs. AFLGo-all denote AFLGo that targets all three bugs, and AFLGo-B1, AFLGo-B2, AFLGo-B3 denote AFLGo that only targets one of the bugs, Bug 1, Bug 2, and Bug 3, respectively. The average time taken to find the bug (TTE) and the number of runs that found the bug within the 24 hours of fuzzing are reported for each fuzzer.

Measure	**Fuzzer**	**20×20**			**30×30**		
		Bug 1	**Bug 2**	**Bug 3**	**Bug 1**	**Bug 2**	**Bug 3**
TTE (h)	**AFLGo-all**	0.79	0.51	0.71	2.23	3.51	3.00
	AFLGo-B1	0.64	0.37	0.55	2.81	4.73	3.51
	AFLGo-B2	0.52	0.36	0.53	1.24	1.42	1.58
	AFLGo-B3	0.45	0.35	0.43	1.32	2.70	1.38
# of hits over 10 runs	**AFLGo-all**	10	10	10	10	10	10
	AFLGo-B1	10	10	10	10	9	10
	AFLGo-B2	10	10	10	10	10	10
	AFLGo-B3	10	10	10	10	10	10

F. RQ5. Impact of Multiple Targets

We now evaluate the effectiveness of targeting multiple locations in DGF with two synthesized programs of sizes 20x20 and 30x30, each with *three* bugs. Note that we used the modified Fuzzle to synthesize bugs of the same difficulty, that is, the bugs with the same *solution length* in the maze (see §III-A). Note that we only tested AFLGo because Beacon and DAFL do not support fuzzing with multiple targets.

For each program, we ran AFLGo in four different configurations, each with a different set of target locations. Specifically, we had one configuration, denoted by AFLGo-all, where all three bugs in the programs were provided as the target locations. The remaining three configurations, AFLGo-B1, AFLGo-B2, and AFLGo-B3, were given one of the three bugs as the target location. We performed 10 repeats for all four configurations. The results are summarized in Table V.

From the results in Table V, we see that the effect of directed fuzzing of AFLGo diminishes when it is instructed to target multiple locations. Overall, AFLGo-all took longer to find each of the three bugs when compared to running AFLGo with only the respective bug as the target. For example, in the program generated using the maze of size 20x20, AFLGo-all took 0.79 hour, 0.51 hour, and 0.71 hour to find the Bug 1, Bug 2, and Bug 3, respectively, whereas AFLGo-B1, AFLGo-B2, and AFLGo-B3 each took 0.64 hour, 0.36 hour, and 0.43 hour to find the same bugs, respectively. Similarly, AFLGo-all took longer to find the two of the three bugs (Bug 2 and Bug 3) in the program of size 30x30 when compared to the configurations with only a single target, AFLGo-B2 and AFLGo-B3.

Moreover, AFLGo-all does not provide any advantage in reaching all targets. For example, AFLGo-all would need at least 0.79 hour to find all three bugs in the program of size 20x20, whereas AFLGo-B1, AFLGo-B2, and AFLGo-B3 would require 0.64 hour, 0.53 hour, and 0.45 hour, respectively. All these results are intuitively explained as AFLGo assumes

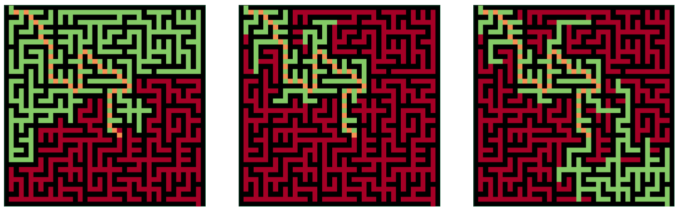
(a) AFLGo (b) Beacon (c) DAFL

Fig. 1: Visualized coverage achievements by AFLGo, Beacon, and DAFL on the program generated using CVE-2016-6131 [28]. Only DAFL reaches the exit of the maze, the target (i.e., bug) location for directed fuzzers.

```
void func_250(char *input, int idx){
  if((uint8_t) 83 == input[idx]){
    func_251(input, idx + 1);
  }
}

void func_251(char *input, int idx){
  if((uint8_t) 84 == input[idx]){
    func_271(input, idx + 1);
  } else {
    func_250(input, idx + 1);
  }
}
```

Fig. 2: Simplified code snippet from the program synthesized using CVE-2016-6131 [28].

that the target locations are relevant to each other, whereas in our experiments, the target locations are irrelevant and far apart from each other (as discussed in §III-A).

Lesson 6. Multi-target directed fuzzing is less effective with multiple targets that are far apart from each other.

G. RQ6. Usefulness of Coverage Visualization

How does the coverage visualization of Fuzzle help understand the bottleneck of directed fuzzers? To answer this question, we visualized the code coverage achieved by fuzzers on the program with constraints from CVE-2016-6131 [28], which is the program that most fuzzers struggled to find the bug in (Table II). Figure 1 shows the visual aids produced by Fuzzle, where the branch conditions from CVEs are highlighted in orange. From Figure 1a and Figure 1b, we can see that the corresponding run of AFLGo and Beacon stopped its progress at the second last and the last branch that use the constraints from the CVE-2016-6131, respectively. In fact, we saw that in most cases, the fuzzers were stuck on either of these last two branches. Specifically, out of 37 total runs that did not find the bug, 24 runs (7 runs of AFL and Beacon, 4 runs of AFL++ and AFLGo, and 2 run of DAFL) failed to pass the second last branch condition and another 13 runs (3 runs of AFL, 4 runs of AFLGo and AFL++, and 1 run of Beacon and DAFL) failed to pass the last branch condition.

Based on the information gathered from the coverage visualization, we identified the two problematic functions, `func_250` and `func_251`, and manually analyzed the source code. Figure 2 is a simplified code snippet of the two functions. By inspecting the code, we can easily see why the fuzzers struggled to pass the last two branches: both branch had equality checks and there was a loop between the two functions containing these branches. In other words, the fuzzers had to generate an input that passes two equality checks on two consecutive input bytes to progress farther in the program. This explains why DAFL significantly outperformed the rest of the fuzzers, including Beacon, in terms of finding bug. As Beacon cannot correctly handle the complex loop structures [20], it does not prune the backward edge from `func_251` to `func_250`. DAFL, on the other hand, uses Def-Use-Graph to prioritize seeds that are semantically relevant to reaching target locations, to successfully find the bug in seven of the ten runs. This example clearly shows that the visualization feature of Fuzzle can help locate and analyze the bottlenecks of fuzzing.

Lesson 7. Coverage visualization of Fuzzle is useful in identifying bottlenecks in directed fuzzing.

V. Discussion and Future Work

In this section, we first address the threats to validity of this study and describe several measures we have taken to mitigate them. We then discuss the implication of using synthetic benchmarks for evaluating directed fuzzers.

A. Threats to Validity

There are several threats to validity of this study: (1) the selection bias for the evaluation targets, (2) the sample sizes, and (3) the representativeness of samples. To mitigate the threats, we have taken several measures when designing our experiments. First, we chose to evaluate multiple directed fuzzers that all employ different techniques for directing the exploration of fuzzers in addition to two widely used general-purpose undirected fuzzers. Additionally, when running the fuzzers, we used their default configurations that are recommended by the authors. To guarantee a sufficient sample size for evaluating directed fuzzers, we created a benchmark comprising 12 programs using Fuzzle. This was achieved by varying the values of different parameters, which additionally ensured a diverse set of benchmark programs. Also, when evaluating the impact of one property, such as program size and the number of bugs, we made sure that the rest of the parameter values are kept constant.

B. Implication of Using Synthetic Benchmarks

While Fuzzle [23] perfectly suits our purpose of evaluating directed grey-box fuzzers, the synthesized bugs from Fuzzle are not necessarily identical to real-world organic bugs. Thus, we do not claim that synthetic bugs can completely replace real-world bugs in terms of evaluating directed fuzzers. However, we have empirically showed that synthetic bugs can be a useful complement to real-world bugs, and they can even help identify the current limitations of DGF techniques as well as the future research directions. Furthermore, synthesizing realistic bugs is an ongoing research problem, and we believe that further progress in this direction will help improve the state of the art of DGF.

VI. Related Work

A. Directed Grey-box Fuzzing

Fuzzing is a de facto standard for testing software systems [25]. It has been used by major software companies such as Google [1], [2] and Microsoft [27] to find thousands of real-world vulnerabilities. Almost every layer of the software stack has been a target for fuzzing, including the operating systems [9], [34], [35], compilers [7], [24], [36], and web browsers [14], [15], [39].

Directed grey-box fuzzing is a variant of fuzzing that has been gaining momentum in recent years due to its power in generating test cases reaching specific target locations in the program under test. AFLGo [3] is the first directed grey-box fuzzer that introduces the notion of seed distance, which is an average distance from executed nodes in the CFG to the target node(s). Hawkeye [6] improves the seed distance mechanism of AFLGo by considering the frequency of call edges in the call graph. WindRanger [11] shows that not every basic block in the CFG is equally important, and proposes a way to select a subset of the basic blocks, known as deviation basic blocks, to compute the seed distance.

There are several recent approaches that tackle other aspects in DGF. Beacon [17] employs a weakest precondition analysis to prune off infeasible paths in the program under test. FuzzGuard [40] leverages deep learning to predict the probability of a test case reaching the target location. MC^2 [33] presents a novel oracle, named noisy counting oracle, which can approximate the likelihood of an input reaching the target location. To construct such an oracle, it uses Monte Carlo sampling with concentration bounds to estimate the upper-bound probability. DAFL [20] analyzes the data dependency of the target location to selectively add instrumentation routines only to the relevant parts of the program under test.

B. Bug Synthesis

Most existing fuzzing benchmarks are manually constructed by collecting real-world buggy programs [16], [18]. But those benchmarks lack the ground truth of the bugs: the locations of the bugs and triggering inputs of the bugs are not known, or, at least, difficult to collect them at scale [4].

Bug synthesis is a technique that addresses such problems by automatically generating buggy programs with known bugs. LAVA [10] is the first bug synthesis tool that generates buggy programs by injecting bugs into random locations of a program, where every injected bug is guarded by a magic value check. However, fuzzers can easily overfit to the benchmarks generated by LAVA by learning the patterns of the magic value checks or by penetrating the equality constraints [8], [21]. EvilCoder [30] injects bugs into a program by removing security checks or input sanitization routines. Apocalypse [31] focuses on injecting deep bugs that are difficult for fuzzers to find. Specifically, it modifies a program in such a way that the injected bug is only triggered when a state machine, named error transition system, reaches a specific state. Bug-Injector [19] leverages bug templates representing previously known bugs to insert bugs into a host program. FixReverter [38] searches for known bugfix patterns and modifies the matched code snippets to introduce bugs in order to ensure the generation of realistic bugs.

All these approaches share a common theme: they insert bugs into an existing program to produce benchmarks. However, program modification may introduce unintended bugs, and existing programs can always contain previously unknown bugs. Therefore, it is extremely difficult to obtain a ground truth of the synthesized benchmarks.

Fuzzle [23] is the first attempt in addressing this problem by generating a whole benchmark from scratch. As it constructs a program by creating a maze of functions, it guarantees that the generated program has a bug at a known location.

The closest work to ours is Bundt *et al.* [4], whose goal is to evaluate the effectiveness of synthetic bugs in evaluating *undirected* fuzzers. To our knowledge, Fuzzle as well as all the other existing bug synthesis tools have only been used to evaluate undirected fuzzers. This paper is the first attempt to leverage a bug synthesis technique for evaluating directed grey-box fuzzers.

VII. Conclusion

In this paper, we conducted a series of experiments with synthetic programs generated by Fuzzle to systematically understand the effectiveness of DGF. As a result, we gathered several useful insights that would have been difficult to obtain with traditional benchmarks. First, we found that synthetic benchmarks give us a clear understanding of the limitations of existing DGF techniques. For example, we found that DGF tools struggle to reach target bugs guarded with complex path conditions. Second, we found that synthetic benchmarks enable us to identify the exact bottleneck of a fuzzer. Finally, we found that DGF is less affected by the size of the benchmark program than undirected fuzzing, which is indeed a desirable property for DGF. We trust that our findings will enable researchers to develop more effective DGF techniques in the future.

VIII. Acknowledgements

We thank anonymous reviewers for their feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, “Announcing OSS-Fuzz: Continuous fuzzing for open source software,” Google Testing Blog, 2016.

[2] M. Böhme, V. J. M. Manès, and S. K. Cha, “Boosting fuzzer efficiency: An information theoretic perspective,” in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2020, pp. 678–689.

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[4] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, “Evaluating synthetic bugs,” in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2021, pp. 716–730.

[5] S. Canakci, N. Matyunin, K. Graffi, A. Joshi, and M. Egele, “Targetfuzz: Using darts to guide directed greybox fuzzers,” in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 561–573.

[6] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a desired directed grey-box fuzzer,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[7] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, “Taming compiler fuzzers,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

[8] J. Choi, J. Jang, C. Han, and S. K. Cha, “Grey-box concolic testing on binary code,” in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[9] J. Choi, K. Kim, D. Lee, and S. K. Cha, “NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 1973–1989.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “LAVA: Large-scale automated vulnerability addition,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2016, pp. 110–121.

[11] Z. Du, Y. Li, Y. Liu, and B. Mao, “Windranger: A directed greybox fuzzer driven by deviation basic blocks,” in *Proceedings of the International Conference on Software Engineering*, 2022, pp. 2440–2451.

[12] E. Falkenauer, “On method overfitting,” *Journal of Heuristics*, vol. 4, pp. 281–287, 1998.

[13] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++ : Combining incremental steps of fuzzing research,” in *Proceedings of the USENIX Workshop on Offensive Technologies*, 2020.

[14] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, “Fuzzilli: Fuzzing for javascript JIT compiler vulnerabilities,” in *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[15] H. Han, D. Oh, and S. K. Cha, “CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines,” in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[16] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[17] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed grey-box fuzzing with provable path pruning,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2022, pp. 36–50.

[18] G. Inc., “fuzzer-test-suite,” https://github.com/google/fuzzer-test-suite, 2016.

[19] V. Kashyap, J. Ruchti, L. Kot, E. Turetsky, R. Swords, S. A. Pan, J. Henry, D. Melski, and E. Schulte, “Automated customized bug-benchmark generation,” in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, 2019, pp. 103–114.

[20] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, “DAFL: Directed grey-box fuzzing guided by data dependency,” in *Proceedings of the USENIX Security Symposium*, 2023, pp. 4931–4948.

[21] lafintel, “Circumventing fuzzing roadblocks with compiler transformations,” https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[22] G. Lee, W. Shim, and B. Lee, “Constraint-guided directed greybox fuzzing,” in *Proceedings of the USENIX Security Symposium*, 2021, pp. 3559–3576.

[23] H. Lee, S. Kim, and S. K. Cha, “Fuzzle: Making a puzzle for fuzzers,” in *Proceedings of the International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[24] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, “Many-core compiler fuzzing,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[25] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,”

IEEE Transactions on Software Engineering, vol. 47, no. 11, pp. 2312–2331, 2021.

[26] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An open fuzzer benchmarking platform and service," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2021, pp. 1393–1403.

[27] Microsoft, "OneFuzz: A self-hosted fuzzing-as-a-service platform," https://github.com/microsoft/onefuzz.

[28] MITRE Corporation, "CVE-2016-6131," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-6131.

[29] MITRE Corporation, "CVE-2016-9827," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-9827.

[30] J. Pewny and T. Holz, "EvilCoder: Automated bug insertion," in *Proceedings of the Annual Computer Security Applications Conference*, 2016, pp. 214–225.

[31] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2018, pp. 224–234.

[32] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[33] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, and S. Jana, "MC^2: Rigorous and efficient directed greybox fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2022, pp. 2595–2609.

[34] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "KSG: Augmenting kernel fuzzing with system call specification generation," in *Proceedings of the USENIX Annual Technical Conference*, 2022, pp. 351–366.

[35] D. Vyukov, "syzkaller," https://github.com/google/syzkaller.

[36] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "JITfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *Proceedings of the International Conference on Software Engineering*, 2023.

[37] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[38] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FixReverter: A realistic bug injection methodology for benchmarking fuzz testing," in *Proceedings of the USENIX Security Symposium*, 2022, pp. 3699–3715.

[39] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: Browser API fuzzing with dynamic mod-ref analysis," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2022, pp. 1135–1147.

[40] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 2255–2269.