Received: 10 January 2023 | Revised: 27 September 2023 | Accepted: 12 November 2023

DOI: 10.1002/stvr.1869

SURVEY ARTICLE

WILEY

The progress, challenges, and perspectives of directed greybox fuzzing

Pengfei Wang | **Xu Zhou** | **Tai Yue** | **Peihong Lin** | **Yingying Liu** | **Kai Lu**

College of Computer, National University of Defense Technology, Changsha, China

Correspondence
Pengfei Wang, College of Computer, National University of Defense Technology, No.109 Deya Road, Kaifu District, Changsha, China.
Email: pfwang@nudt.edu.cn

Funding information
National University of Defense Technology Research Project, Grant/Award Numbers: ZK20-17, ZK20-09; National Natural Science Foundation China, Grant/Award Numbers: 62272472, U22B2005, 61972412; HUNAN Province Natural Science Foundation, Grant/Award Number: 2021JJ40692

Summary
Greybox fuzzing is a scalable and practical approach for software testing. Most greybox fuzzing tools are coverage-guided as reaching high code coverage is more likely to find bugs. However, since most covered codes may not contain bugs, blindly extending code coverage is less efficient, especially for corner cases. Unlike coverage-guided greybox fuzzing which increases code coverage in an undirected manner, directed greybox fuzzing (DGF) spends most of its time allocation on reaching specific targets (e.g. the bug-prone zone) without wasting resources stressing unrelated parts. Thus, DGF is particularly suitable for scenarios such as patch testing, bug reproduction, and special bug detection. For now, DGF has become an active research area. However, DGF has general limitations and challenges that are worth further studying. Based on the investigation of 42 state-of-the-art fuzzers that are closely related to DGF, we conducted the first in-depth study to summarize the empirical evidence on the research progress of DGF. This paper studies DGF from a broader view, which takes into account not only the location-directed type that targets specific code parts but also the behavior-directed type that aims to expose abnormal program behaviors. By analyzing the benefits and limitations of DGF research, we try to identify gaps in current research, meanwhile, reveal new research opportunities and suggest areas for further investigation.

KEYWORDS
directed greybox fuzzing, location directed fuzzing, target directed fuzzing

1 | INTRODUCTION

Fuzzing is an automated software testing approach proposed by Barton Miller in 1989 [1]. By repeatedly and randomly mutating the inputs to the program under test (PUT), fuzzing is effective and practical in vulnerability detection. As one of the most efficient and scalable fuzzing categories, greybox fuzzing [2–7] has developed rapidly in recent years. Based on the feedback information from the execution of the PUT, greybox fuzzers use an evolutionary algorithm to generate new inputs and explore paths. Greybox fuzzing is widely used to test application software, libraries [8, 9], kernel code [10–12], and protocols [13–15]. Most greybox fuzzing tools are coverage-guided, which aim to cover as many program paths as possible within a limited time budget. This is because, intuitively, reaching high code coverage is more likely to find bugs. However, it is not appropriate to treat all codes of the program as equal because most covered codes may not contain bugs. For example, according to Shin et al. [16], only 3% of the source code files in Mozilla Firefox have vulnerabilities. Thus, testing software by blindly extending code coverage is less efficient, especially for corner cases. Since achieving full code coverage is difficult in practice, researchers have been trying to target the vulnerable parts of a program to improve efficiency and save resources. Directed fuzzing is proposed as a means of achieving this aim [17].

Pengfei Wang and Xu Zhou contributed equally to this work and should be regarded as co-first authors on this work.

Unlike coverage-based fuzzers that blindly increase the path coverage, a directed fuzzer focuses on target locations (e.g. the bug-prone zone) and spends most of its time budget on reaching these locations without wasting resources stressing unrelated parts. Originally, directed fuzzers were based on symbolic execution [17–20], which uses program analysis and constraint solving to generate inputs that exercise different program paths. Such directed fuzzers cast the reachability problem as an iterative constraint satisfaction problem [21]. However, since directed symbolic execution (DSE) relies on heavy-weight program analysis and constraint solving, it suffers from scalability and compatibility limitations.

In 2017, Böhme et al. introduced the concept of directed greybox fuzzing (DGF) [21]. Greybox fuzzing generates inputs by mutating seeds. By specifying a set of target sites in the PUT and leveraging lightweight compile-time instrumentation, a directed greybox fuzzer can use the distance between the input and the target as the fitness function to assist seed selection. By giving more mutation chances to seeds that are closer to the target, it can steer the greybox fuzzing to reach the target locations gradually. Unlike traditional fuzzing techniques, DGF casts reachability as an optimization problem whose aim is to minimize the distance between generated seeds and their targets [21]. Compared with DSE, DGF has much better scalability and improves efficiency by several orders of magnitude. For example, Böhme et al. can reproduce the Heartbleed vulnerability within 20 min, while the DSE tool KATCH [20] needs more than 24 hours [21].

Motivation. For now, DGF has become a research hot spot and it is growing very fast. It has evolved beyond the original pattern that depends on manually labeled target sites and distance-based metrics to prioritize the seeds. New fitness metrics, such as trace similarity and vulnerability prediction models, are used. Current DGF tools can not only identify targets automatically but also expose target program behavior in a directed manner. A great number of variations have been used to boost software testing under different scenarios, such as patch testing [22–24], regression testing [25, 26], bug reproduction [24, 27, 28], knowledge integration [29], result validation [30–33], energy-saving [15] and special bug detection [15, 24, 34–38]. Though fast-growing and useful, DGF has general limitations and challenges that are worth further study. Against this background, we conduct this work to summarize the empirical evidence on the research progress of DGF. Based on the analysis of the benefits and limitations of DGF research, we try to identify gaps in current research, meanwhile, reveal new research opportunities, and suggest areas for further investigation.

Research questions. We conduct the first in-depth study of DGF in this work. To study DGF from a broader view, we take into account not only the location-directed type that targets specific code parts but also the behavior-directed type that targets exposing abnormal program behaviors to find bugs. In summary, we design the following research questions:

- **RQ1**: How the target identification method is changed in the up-to-date research of DGF?
- **RQ2**: In addition to distance, are there any new fitness metrics in the recent development of DGF?
- **RQ3**: How the recent DGF tools are optimized regarding the key steps of fuzzing?
- **RQ4**: What are the challenges of the DGF research? Are there any potential solutions?
- **RQ5**: What are the typical applications of DGF? How to choose a DGF tool for a specific application scenario?
- **RQ6**: What are the perspectives on the future trends in DGF research?

In this work, we make the following contributions.

- We investigate 42 state-of-the-art fuzzers that are closely related to DGF to systemize recent progress in the field and answer research questions **RQ1**, **RQ2,** and **RQ3**.
- Based on the analysis of the known works, a summary of five challenges to DGF research is provided. We discuss these challenges with examples and disclose the deep reasons behind them, aiming to propose possible solutions to address them and answer **RQ4**.
- Based on the fast-growing rate of DGF tools, we summarize the typical application scenarios of DGF and provide suggestions on how to choose a DGF tool for a specific application scenario, which answers **RQ5**.
- We make suggestions in terms of the perspectives for the research points of DGF that are worth exploring in the future, aiming to facilitate and boost research in this field and answer **RQ6**.

2 | BACKGROUND

2.1 | Blackbox fuzzing

Blackbox fuzzing is the original and simplest form of fuzzing [1]. The workflow of blackbox fuzzing is represented by the blue long dashed lines in Figure 1. It randomly mutates the inputs and then tests the PUT with these modified

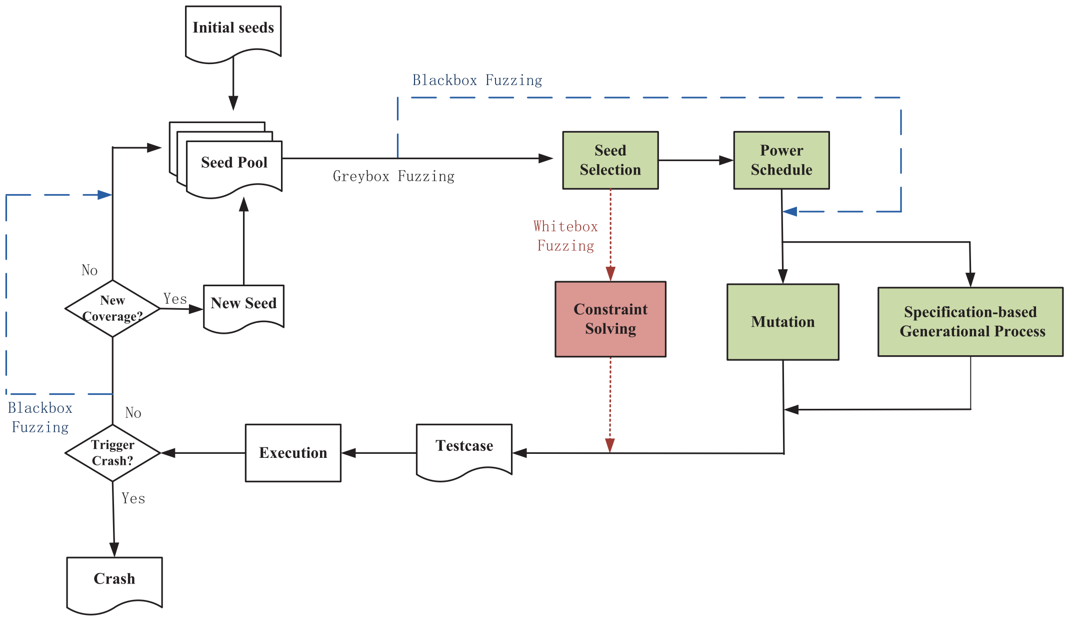


FIGURE 1 Workflow of different fuzzing approaches.

inputs. The procedure can be repeatedly executed to generate as many new inputs as needed. However, since blackbox fuzzing does not support any feedback scheme, the inputs are generated blindly. Thus, the code coverage is usually low, and most of the generated inputs are invalid or redundant, resulting in low testing efficiency. Though simple and less efficient, blackbox fuzzing can be effective in finding security vulnerabilities, especially for testing scenarios where the feedback scheme of greybox fuzzing is difficult to realize, such as protocol testing [13, 39–42].

2.2 | Coverage-guided greybox fuzzing (CGF)

CGF aims to maximize the code coverage to find hidden bugs. By introducing a feedback scheme, the fuzzing status (i.e. whether a new path is explored) can be leveraged to guide the generation of new inputs. Figure 1 shows the workflow of CGF, where the components shown in the white boxes are the common steps used by almost all the fuzzing approaches, while the components shown in the green boxes are critical steps for CGF. The testcase in CGF can be generated in a mutational or generational way, and a typical mutational CGF includes the following steps:

- **Seed selection**. Select a proper seed (usually considered as high quality) from the seed queue for mutation;
- **Power schedule**. Determine the energy (i.e. the number of mutation chances) assigned to the selected seed;
- **Mutation**. Mutate the seed with a set of pre-defined mutation strategies to generate test inputs;
- **Execution**. Run the PUT with the test inputs to monitor whether a new path is exercised. If a new path is triggered, the input is added to the queue as a new seed.

Here we take the widely used tool AFL (American fuzzy lop) [43] as a representative to illustrate the principle of CGF. AFL is a prevalently used coverage-based greybox fuzzer, and many state-of-the-art greybox fuzzers [44–48] are built on top of it. AFL uses lightweight instrumentation to capture basic block transitions at compile-time and gain coverage information during run-time. It then selects a seed from the seed queue and mutates the seed to generate test cases. If a test case exercises a new path, it is added to the queue as a new seed. AFL favors seeds that trigger new paths and gives them preference (i.e. more energy) over the nonfavoured ones. Compared with other instrumented fuzzers, AFL has a modest performance overhead. However, though some tools (e.g. Fairfuzz [49]) try to reach the rare part of a code, most greybox fuzzers treat all codes of the program as equal. Thus, CGF is less efficient as effort is wasted on nonbuggy areas.

Algorithm 1: Directed greybox fuzzing scheme.

```
Input:  $i$ – Initial input
Input:  $Target$ – A set of target locations
Output: $BugInput$ – A set of buggy input
01  $BugInput \leftarrow \varnothing$
02  $SeedQueue \leftarrow i$
03  while true do
04    $s \leftarrow select(SeedQueue)$
05    $s' \leftarrow mutation(s)$
06    $trace \leftarrow execution(s')$
07    if $find\_new\_path(trace)$ then
08      $SeedQueue \leftarrow SeedQueue + s'$
09    if $trigger\_crash(trace)$ then
10      $BugInput \leftarrow BugInput + s'$
11    $distance \leftarrow evaluate\_seed(trace, Targets)$
12    $SeedQueue \leftarrow sort\_insert(SeedQueue, s', distance)$
13  end
```

2.3 | DGF

Unlike CGF which blindly increases path coverage, DGF aims to reach a set of pre-identified locations in the code (potentially the buggy parts) and spends most of its time budget on reaching target locations without wasting resources stressing unrelated parts. To describe the DGF principle, we use AFLGo [21] as an example. AFLGo follows the same general principles and architecture as CGF. At compile-time, except for instrumentation, AFLGo also calculates the distances between the input and predefined targets. The distance is calculated as the average weight of the execution trace to the target basic blocks. The execution trace weight is determined by the number of edges in the call graph and control-flow graphs of the program. Then, at run-time, AFLGo [21] prioritizes seeds based on distance instead of new path coverage and gives preference to seeds closer to the targets at basic block level distance. Böhme et al. [21] view the greybox fuzzing process as a Markov chain that can be efficiently navigated using a "power schedule". They leveraged a simulated annealing strategy to gradually assign more energy to a seed that is closer to targets than to a seed that is further away. They cast reachability as an optimization problem to minimize the distance between the generated seeds and their targets [44]. Similar to CGF, in Figure 1, the components shown in the green boxes are also critical steps for DGF, and DGF is mainly optimized via these steps.

The exploration–exploitation problem. DGF fuzzing is a two-part method, which is readily separated into phases of *exploration* and *exploitation* [21]. The exploration phase is designed to uncover as many paths as possible. Like many coverage-guided fuzzers, DGF in this phase favors seeds that trigger new paths and prioritizes them. This is because new paths increase the potential to lead to targets, and it is particularly necessary when the initial seeds are quite far from their targets. Then, based on the known paths, the exploitation phase is invoked to drive the engine to the target code areas. In this phase, Böhme et al. [21] prioritize seeds that are closer to the targets and assign more energy to them. The intuition behind this is that if the path that the current seed executes is closer to any of the expected paths that can reach the target, more mutations on that seed should be more likely to generate expected seeds that fulfill the demands. The exploration–exploitation trade-off lies in how to coordinate these two phases. Böhme et al. [21] use a fixed splitting of the exploration and exploitation phases. For example, in a 24-hour test, AFLGo assigns 20 hours for exploration and then 4 hours for exploitation.

2.4 | Directed whitebox fuzzing

A directed whitebox fuzzer [50] is mostly implemented into a symbolic execution engine such as KLEE [51], KATCH [20] and BugRedux [52]. DSE uses program analysis and constraint solving to generate inputs that systematically and effectively explore the state space of feasible paths [18]. In Figure 1, the constraint-solving component in the red box is mandatory for whitebox fuzzing but optional for DGF. Once a target path is identified, potential solutions to the path constraints are explored by creating test cases. Since most paths are actually infeasible, the search usually proceeds iteratively by finding feasible paths to intermediate targets. Unlike DGF, which casts reachability as an optimization problem to minimize the distance between generated seeds and their targets [21], DSE casts the reachability problem as an iterative constraint satisfaction problem [21]. DSE is effective in various scenarios, such as reaching error-prone program locations (e.g. critical syscalls [53]), testing code patches [20, 54, 55], exercising corner paths to increase coverage [56], and reproducing failures in-house [52, 57].

However, DSE's effectiveness comes at the cost of efficiency. The heavy-weight program analysis and constraint solving of DSE are rather time-consuming. At each iteration, DSE utilizes program analysis techniques to identify branches that can be negated to get closer to the target. Then, based on the sequence of instructions along these paths, it constructs the corresponding path conditions. Finally, it checks the satisfiability of those conditions using a constraint solver. DGF is capable of producing a far greater number of inputs in a given timeframe than DSE can achieve [21]. Böhme et al. have demonstrated with experiments that DGF outperforms DSE both in terms of effectiveness and efficiency. For example, AFLGo can expose the Heartbleed vulnerability in 20 min, while the DSE tool KATCH cannot even in 24 hours [21].

2.5 | Search-based software testing (SBST)

SBST formulates a software testing problem into a computational search problem that can be optimized with meta-heuristic search techniques, such as hill-climbing, simulated annealing, and genetic algorithms [58]. The key to the optimization process is defining a problem-specific fitness function, which guides the search by measuring the quality of potential solutions from a possibly infinite search space. Greater fitness values are assigned to those inputs that provide data closer to the focal point in the program [59]. The original use of SBST was structural coverage testing [60], including path and branch coverage. The path taken through the PUT is compared with some structure of interest for which coverage is sought [59]. The fitness function usually incorporates two metrics—approach level and branch distance [61]. The complete fitness value is computed by normalizing the branch distance and adding it to the approach level [59]. In addition to structural testing, SBST can also be used for functional testing [62, 63], temporal testing [64–66], robustness testing [67], integration testing [68, 69], regression testing [70], stress testing [71], mutation testing [72], interaction testing [73–75], state testing [76–78] and exception testing [79, 80]. The main difference between fuzzing and SBST is that fuzzing uses lightweight and scalable heuristics to maintain and evolve a population of test inputs, whereas SBST approaches typically use an optimization formula to search for ideal test cases [81].

Though SBST and DGF are close, they are different. For example, fuzzing uses lightweight instrumentation to guide the evolution; thus, it is simpler than the computational search optimization in SBST. Besides, the search optimization of SBST might be trapped by the local optimal solution, while DGF can escape the local optimal solution more quickly due to the randomness of fuzzing. In this paper, we focus on DGF and provide the most up-to-date research progress.

3 | METHODOLOGY

This section introduces the methodology we adopted when conducting this research. The motivation and research questions have been introduced in Section 1; thus, here, we only describe the other key elements in a review protocol.

3.1 | Inclusion and exclusion criteria

This paper defines a tool as a directed greybox fuzzer from a broader view, namely, that a fuzzer either reaches specific target locations or triggers specific program buggy behavior by optimizing a customized fitness function. The following inclusion criteria are thus specified, which also serve as the definition of DGF in this paper.

- The core mechanism should be greybox fuzzing, which relies on the instrumentation of the PUT and includes the key steps of seed prioritization, power scheduling, and mutator scheduling.
- The directedness is realized by optimizing the fitness metric in the key steps of greybox fuzzing, which includes input optimization, seed prioritization, power scheduling, mutator scheduling and mutation operations.
- The fitness goal is to reach a specific site or to trigger certain buggy behavior of a program. The site could be a manually labeled target or a potential bug location predicted automatically, such as by machine learning [30–32] or static analysis [33]. The target buggy behavior could be a nonfunctional property (e.g. memory consumption [34]) or a certain bug type (e.g. algorithmic complexity vulnerability [37]).

We classify a DGF tool as directed for target location type when its object is reaching target sites, and the fitness metric can be measured visibly on a concrete structure, such as on the execution trace, the control-flow graph, or the call-graph. The target can be a single location, a set of basic blocks, or a sequence of ordered call sites. In contrast, if a DGF tool is directed with a certain fitness metric but without a fixed target, then it is classified as directed for target behavior. For this type, the targets need not or cannot be prelabelled, and the fitness metric is not as visible as the first type. With the optimization of the fitness function, a target can be reached automatically and the buggy behavior will be exposed.

However, to concentrate on the research of DGF, the following types of papers will be excluded.

- Directed whitebox fuzzing realized only via symbolic execution (we still include directed hybrid fuzzing papers that assist DGF with symbolic execution)
- Papers on SBST
- Informal literature reviews and technical reports
- Too short papers (less than four pages) without a clear description of the approach or the evaluation.

3.2 | Search process

The search process consists of three rounds. The first round is a manual search of specific conference proceedings and journal papers via an academic search engine with keywords, which includes the following steps.

(1) The publications are initially collected from the proceedings of the top-level conferences on security and software engineering since 2017. Alphabetically, ACM Conference on Computer and Communications Security (CCS), IEEE Symposium on Security and Privacy (S&P), USENIX Security Symposium (Sec), Network and Distributed System Security Symposium (NDSS) and International Conference on Software Engineering (ICSE), ACM International Symposium on the Foundations of Software Engineering ESEC/FSE) and IEEE/ACM International Conference on Automated Software Engineering (ASE). We searched with “DGF” and “directed fuzzing”. We collected 19 papers.
(2) Then, we used Google Scholar to search for works from journals and preprints by searching with keywords including “DGF”, “directed fuzzing” and “targeted fuzzing”. We collected 15 papers.
(3) After that, we referred to a popular fuzzing paper repository[2] and manually selected papers related to DGF. We also refer to another paper repository that only collects papers related to directed fuzzing[3]. We collected 21 papers.

Then, in the second round, we filter out the duplicates from the collection in the previous round. When a paper has been published in more than one journal/conference, the most complete version will be used. As a result, 49 papers remained.

In the third round, we read each paper we collected and filtered out the papers based on the research content with the inclusion and exclusion criteria from Section 3.1. Finally, 42 papers ranging from 2017.1 to 2022.5 (listed in Table 1) remained for further investigation.

3.3 | Data collection

First, we interviewed at least 10 researchers to list the important aspects of DGF tools they care about. The researcher includes postgraduate students, PhD students, and faculties. Then, we summarized the suggestions and extracted the aspects that they care about most. Based on this, the data extracted from each paper will be:

- The publication source (i.e. the conference, journal, or preprint) and year.
- The fitness goal. To reach what kind of target sites (e.g. vulnerable function) or to expose what target bugs?
- The fitness metric used in the evolutionary process of fuzzing. For example, the distance to the targets.
- How the targets are identified or labeled? For example, predicted by deep learning models.
- The implementation information. What tool is the fuzzer implemented based on? Is the fuzzer open-sourced?
- Whether the tool support binary code analysis?
- Whether the tool support kernel analysis?
- Whether the tool support multi-targets searching?
- Whether the tool support multi-objective optimization?

TABLE 1 Collection of directed greybox fuzzers.

Category	Tools	Publication	Fitness goal	Fitness metric	Target identify	Base tool	Binary support	Kernel support	Open sourced	Multi-targets	Multi-objective
Directed for target location	AFLGo [21]	CCS'17	Target sites	Distance	Manual label	AFL	×	×	✓	✓	×
	SemFuzz [22]	CCS'17	Target function/site	Distance	Automatic by NLP	Syzkaller	×	✓	×	✓	×
	Hawkeye [82]	CCS'18	Target site	Distance	Manual label	AFL	×	×	×	✓	×
	LOLLY [83]	ICPC'19	Target sequence	Sequence coverage	Manual label	AFL	×	×	×	✓	×
	TAFL [84]	ICSE'19	Vulnerable region	Customized path weights	Static semantic analysis	AFL	×	×	✓	✓	×
	DrillerGo [28]	CCS'19	Vulnerable function	Coverage	Manually based on CVE info	AFL	Angr	×	×	×	×
	1DVUL [23]	DSN'19	Binary patches	Distance	Binary diffing	Driller	QEMU	×	×	✓	×
	Wüstholz [85]	Arxiv'19	Target sites	Path reachability	Static analysis	HARVEY	BRAN	×	×	✓	×
	SUZZER [31]	ICISC'19	Vulnerable function	Vulnerable probability	Predict by deep learning	VUzzer	IDA	×	×	✓	×
	V-Fuzz [30]	TCM'20	Vulnerable function	Vulnerable probability	Predict by deep learning	VUzzer	IDA	×	×	✓	×
	DeFuzz [32]	Arxiv'20	Vulnerable location	Vulnerable probability	Predict by deep learning	AFLGo	×	×	×	✓	×
	AFLPro [86]	JISA'20	Sanity checks	Multi-dimensional fitness	Automatic	AFL	QEMU	×	×	✓	✓
	TortoiseFuzz [27]	NDSS'20	Vulnerable function	Sensitive edge hit rate	Manually based on CVE info	AFL	×	×	×	✓	×
	Berry [87]	SANER'20	Target sequence	Execution trace similarity	Static analysis	AFL	×	×	×	✓	×
	RDFuzz [88]	MPE'20	Target sites	Distance, frequency	Manual label	AFL	×	×	×	✓	×
	TOFU [89]	Arxiv'20	Target sites	Distance	Manual label	-	×	×	×	✓	×
	GTFuzz [90]	PRDC'20	Guard tokens	Distance	Static analysis	AFLGo	×	×	×	✓	×
	ParmeSan [33]	Sec'20	Sanitizer checks	Distance	Static analysis	Angora	×	×	✓	✓	×
	UAFuzz [24]	RAID'20	Use-after-free	Sequence coverage;	Automatic	AFL	QEMU	×	×	✓	×

(Continues)

TABLE 1 (Continued)

Category	Tools	Publication	Fitness goal	Fitness metric	Target identify	Base tool	Binary support	Kernel support	Open sourced	Multi-targets	Multi-objective
	UAFL [35]	ICSE'20	Use-after-free	Operation sequence Coverage	Automatic	AFL	×	×	×	✓	×
	FuzzGuard [91]	Sec'20	Target sites	Distance	Manual label	AFLGo	×	×	×	✓	×
	BEACON [92]	S&P'21	Target sites	Distance	Manual label	AFLGo	×	×	✓	✓	×
	CAFL [93]	Sec '21	Target sites	Conditions to the target	Manual label	AFL	×	×	✓	✓	×
	AFLChurn [26]	CCS '21	Target sites	Distance	All commits	AFL	×	×	✓	✓	×
	DeltaFuzz [25]	JCST '21	Target sites	Distance	Change point	AFL	×	×	×	✓	×
	DirectFuzz [94]	DAC'21	Target sites	Distance	Manual label	AFL	×	×	✓	✓	×
	DGF-CFG Constructor [95]	MDPI'21	Target sites	Distance	Indirect jump	AFLGo	×	×	×	✓	×
	KCFuzz [96]	ICAIS'21	Target sites	Keypoint coverage	Static analysis	AFLGo	×	×	×	×	×
	WindRanger [97]	ICSE'22	Target sites	Distance	Static analysis	AFL	×	×	×	✓	×
Directed for target behaviour	SlowFuzz [38]	CCS'17	Algorithmic Complexity Vulnerability	Resource usage	Automatic	LibFuzzer	×	×	×	×	×
	PERFFUZZ [37]	ISSTA'18	Algorithmic Complexity Vulnerability	Coverage and edge hit count	Automatic	AFL	×	×	✓	✓	✓
	TIFF [98]	ACSAC'18	Buffer overflow, Integer overflow	New coverage	Manual label	VUzzer	×	×	×	×	×
	Joffe [9]	ICST'19	Crash	Crash likelihood	Identified by machine learning	AFL	×	×	×	✓	×
	FuzzFactory [99]	OOPSLA'19	Domain-specific goal	Domain-specific Multi-dimensional objectives	Automatic	AFL	×	×	✓	×	✓
	RVFUZZER [36]	Sec'19	Input validation bug	Control instability	Automatic	-	✓	×	×	×	×
	SAVIOR [100]	S&P'20	Out-of-boundary, Integer overflow, Oversized shift	Bug potential coverage	Annotate by UBSan	AFL	×	×	×	✓	×
	AFL-HR [81]	ICSEW'20	Buffer overflow, Integer overflow	Coverage and headroom	Automatic	AFL	×	×	×	✓	✓

TABLE 1 (Continued)

Category	Tools	Publication	Fitness goal	Fitness metric	Target identify	Base tool	Binary support	Kernel support	Open sourced	Multi-targets	Multi-objective
	GREYHOUND [15]	TDSC'20	Vulnerable behaviour	Multi-dimensional cost functions	Manually by CVE report	AFL	×	×	×	✓	✓
	Memlock [34]	ICSE'20	Memory consumption Bug	Memory usage and path Coverage	Automatic	AFL	×	×	×	✓	✓
	IJON [29]	S&P'20	Deep stateful bug	Path coverage	Human annotation	AFL	×	×	✓	✓	×
	HDR-Fuzz [101]	Arxiv '21	Buffer overrun	Coverage and headroom	ASAN	AFL	×	×	×	✓	✓
	MDPERFFUZZ [102]	ASE'21	Algorithmic Complexity Vulnerability	Coverage and edge hit count	Automatic	PERFFUZZ	×	×	×	✓	✓

- What key steps in fuzzing are optimized to realize the directedness? Namely, input optimization, seed prioritization, power scheduling, mutator scheduling, and mutation operations.
- What techniques are adopted in the optimization? Namely, control-flow analysis, static analysis, data-flow analysis, machine learning, semantic analysis, and symbolic execution.

3.4 | Data analysis

The extracted data are tabulated (Table 1) to show the basic information about each study. Then we review the extracted data and try to answer the research questions as follows:

RQ1: How is the target identification method changed in the up-to-date research on DGF? This will be addressed by summarizing how the targets of the documented research are identified or labeled.
RQ2: In addition to distance, are there any new fitness metrics in the recent development of DGF? This will be addressed by summarizing the fitness metrics of the documented research.
RQ3: How are the recent DGF tools optimized regarding the key steps of fuzzing? This will be addressed by analyzing the documented research on the optimization of the key steps of fuzzing (i.e. input optimization, seed prioritization, power scheduling, mutator scheduling, and mutation operations) to realize the directedness.
RQ4: What are the challenges of DGF research? Are there any potential solutions? We will summarize comprehensive challenges for the DGF community based on the design and implementation of the documented research. For the design, we consider the fitness goal, fitness metric, how the targets are identified, and optimizations on the key fuzzing steps, while for implementation, we pay attention to efficiency and whether the tool supports binary, kernel, multi-targets, and multi-objectives.
RQ5: What are the typical applications of DGF? How to choose a DGF tool for a specific application scenario? We will summarize the typical application of DGF based on the fitness goals, how the targets are identified, and the implementation details of the documented research.
RQ6: What are the perspectives and the future trends on DGF research? We will summarize future trends based on the analysis of the challenges and limitations of the current DGF research.

4 | RESEARCH PROGRESS ON DGF

4.1 | Overview

Recently, DGF has been an active research area. To provide an overview of the DGF research, we summarize the following progress.

- In addition to the original fitness metric of distance, new fitness metrics have been adopted, such as sequence coverage, which is suitable for satisfying complex bug-triggering paths. Examples include UAFuzz [24], UAFL [35], LOLLY [83], Berry [87] and CAFL [93]. Multi-dimensional fitness metrics are also proposed to detect hard-to-manifest vulnerabilities. Examples include AFL-HR [81], HDRFuzz [101] and AFLPro [86].
- To facilitate target identification, tools based on machine learning can predict and label potential targets automatically, examples include SUZZER [31], V-Fuzz [30], DeFuzz [32], and SemFuzz [22]. Meanwhile, CVE information, commit changes, binary diffing techniques, and tools such as UBSan and AddressSanitizer, are adopted to label various potential vulnerable code regions. Examples include DrillerGo [28], TortoiseFuzz [27], AFLChurn [26], GREYHOUND [15], DeltaFuzz [25], 1DVUL [23], SAVIOR [100] and HDR-Fuzz [101].
- The fuzzing process has been enhanced with various approaches, such as using data-flow analysis and semantic analysis to generate valid input, using symbolic execution to pass complex constraints. Examples include TOFU [89], TIFF [98], SemFuzz [22], KCFuzz [96], 1DVUL [23] and SAVIOR [100].
- More complex algorithms are adopted to enhance directedness, such as ant colony optimization, optimized simulated annealing, and particle swarm algorithm. Examples include AFLChurn [26], LOLLY [83], and GREYHOUND [15];
- To improve DGF efficiency, target unreachable inputs are filtered out in advance to save execution. Examples include FuzzGuard [91] and BEACON [92].
- DGF has been used to detect specific bug types, such as memory consumption bugs, and algorithm complexity bugs. Examples include MemLock [34], SlowFuzz [38], PERFFUZZ [37] and MDPERFFUZZ [102].

However, current DGF research also suffers some limitations, such as overhead deduction, equal-weighted metric bias, inflexible coordination of exploration and exploitation, source code dependence, lack of multi-object optimization, and lack of multi-target coordination. In the following sections, we will discuss the above advantages and disadvantages in detail.

4.2 | Target identification

According to the definition of DGF (described in Section 3.1), the fitness goal of DGF can be divided into two categories: directed for target locations and directed for targeted bugs. Among the tools we investigated, 69% are directed by target locations, and 31% are directed by target bugs.

4.2.1 | Target locations

A barrier to most directed fuzzing strategies is the need for PUT target prelabelling [21, 82, 85, 88, 89]. Manual labeling relies on the prior knowledge of the target sites, such as the line number in the source code or the virtual memory address at the binary level, to label the target and steer the execution to the desired locations. According to our statistics, 11 out of the 29 tools that target for locations need manual target labeling. However, obtaining such prior knowledge is challenging, especially for the binary code. In order to set target sites reasonably and effectively, researchers use auxiliary metadata, such as code changes from git commit logs [22], information extracted from bug traces [24], semantics from CVE vulnerability descriptions [15, 27, 28], or deep learning models [30–32]. Such auxiliary metadata can help identify vulnerable functions [27, 28, 30–32], critical sites [96], syntax tokens [90], sanity checks [33, 100], and patch-related branches [23, 26] in the code and set such vulnerable code parts or sites as targets. Nevertheless, such target identification schemes still rely on additional efforts to process the information and mark the target on the PUT. It is unsuitable when fuzzing a PUT for the first time or when well-structured information is unavailable.

To improve automation, static analysis tools [33, 84, 85, 87, 103, 104] are used to automatically find potentially dangerous areas in the PUT. However, these tools are often specific to the bug types and programming languages used [33]. Another direction leverages compiler sanitizer passes (e.g. UBSan [105]) to annotate potential bugs in the PUT [33, 100] or uses binary-level comparison (e.g. Bindiff [106]) to identify patch-related target branches [23]. Deep-learning methods have been used to predict potentially vulnerable code at both binary [30, 31] and abstract syntax tree level [32]. Finally, attack surface identification components [107] have also been used to identify vulnerable targets for DGF automatically.

4.2.2 | Target bugs

Most of the DGF tools are designed for functional goals, such as AFLGo; they need to label the potential buggy locations as target sites, and the fitness metrics are designed for approaching the target sites. Such a scheme is suitable for detecting memory corruption bugs with obvious crashes, such as overflow bugs. Among the tools we investigated, 81% are designed for functional goals. However, DGF tools can also detect nonfunctional goals. For this purpose, they need not label target sites, and the fitness metrics are designed to trigger such nonfunctional behavior. Among the tools we investigated, 19% are designed for nonfunctional goals. For example, UAFuzz [24] and UAFL [35] leverage target operation sequences instead of target sites to find use-after-free vulnerabilities whose memory operations (e.g. allocate, use, and free memory) must be executed in a specific order. Memlock [34] uses memory usage as the fitness goal to find uncontrolled memory consumption bugs. IJON [29] leverages annotations from a human analyst to overcome significant roadblocks in fuzzing and find deep-state bugs. RVFUZZER [36] targets input validation bugs in robotic vehicles. GREYHOUND [15] directs a Wi-Fi client to exhibit anomalous behaviors that deviate from Wi-Fi protocols. PERFFUZZ [37] generates pathological inputs to trigger algorithmic complexity vulnerabilities [37, 38]. For the type of DGF tools that target specific bug types, since they do not need to label the target in the PUT, the fuzzer can identify and trigger such bugs in an evolutionary way.

4.2.3 | Distribution of different targets

Based on the introduction of target identification in the previous subsections, we use Figure 2 to show the distribution of different targets among the tools we investigated. For each category in Figure 2, the number before the comma

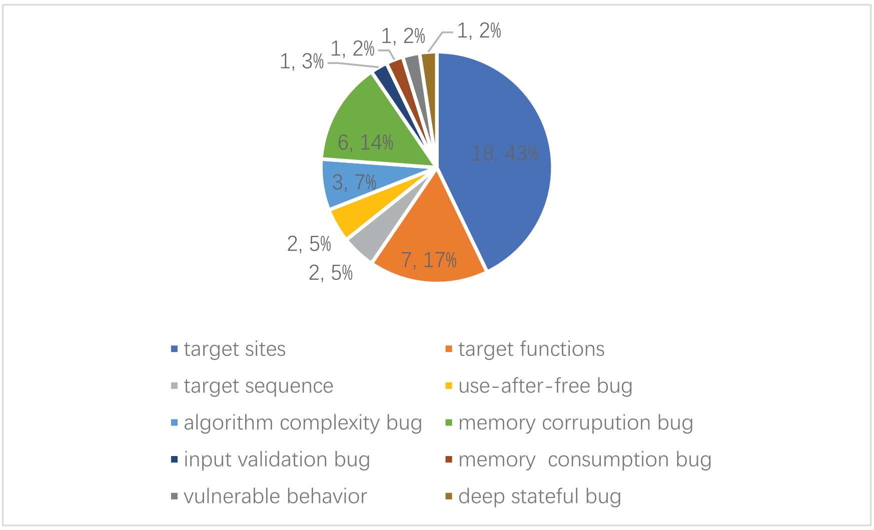


FIGURE 2 Distribution of different targets among the tools we investigated.

indicates the absolute number, while the number after the comma represents the percentage. We can see that tools directed for target sites account for the most (43%), followed by tools directed for target functions, accounting for 17%. For tools directed by bug types, various bug types were considered; among them, memory corruption bug still accounts for the most (14%). Thus, we can conclude that target locations (sites or functions) DGF is still the mainstream research of DGF.

4.3 | Fitness metrics

The crux of DGF is using a fitness metric to measure how the current fuzzing status matches the fitness goal, so as to guide the evolutionary process. We summarize the following fitness metrics used in DGF.

4.3.1 | Distance

Based on our investigation, 31% (13/42) of the directed greybox fuzzers follow the scheme of AFLGo by using the distance between the input and the target as the fitness metric. AFLGo [21] instruments the source code at compile-time and calculates the distances to the target basic blocks by the number of edges in the call and control-flow graphs of the PUT. Then at run-time, it aggregates the distance values of each basic block exercised to compute an average value to evaluate the seed. It prioritizes seeds based on distance and gives preference to seeds that are closer to the target. We use the example in Figure 3 to illustrate how the distance-based metric works. In Figure 3, each node represents a basic block; T1 and T2 are the target basic blocks. The number beside each node indicates the harmonic mean of the distances from each basic block to the target basic blocks. We can calculate the global distances of two execution paths. The distance of path A→B→E→J→O is $(20/9 + 12/7 + 3)/3 \approx 2.31$, while the distance of path A→C→F→K→P→Q is $(20/9 + 4 + 3 + 2 + 1)/5 \approx 2.44$. Since $d_A BEJO < d_A CFKPQ$, the seed that corresponds to path A→B→E→J→O will be prioritized.

Some follow-ups also update this scheme. TOFU's distance metric is defined as the number of correct branching decisions needed to reach the target [89]. RDFuzz [88] combines distance with the execution frequency of basic blocks to prioritize seeds. UAFuzz [24] uses a distance metric of call chains leading to target functions that are more likely to include both allocation and free functions to detect complex behavioral use-after-free vulnerabilities. Different from using equal-weighted basic blocks in the traditional distance calculation, AFLChurn [26] assigns a numerical weight to a basic block based on how recently or how often it has been changed; WindRanger [97] takes into account deviation basic blocks (i.e., basic blocks where the execution trace starts to deviate from the target sites) when calculating distance. One drawback of the distance-based method is that it only focuses on the shortest distance, and thus, longer options might be ignored when there is more than one path reaching the same target, leading to a discrepancy. An example of this problem is depicted in Section 5.3. Another shortcoming is the considerable time cost when calculating the distance at the basic block level. On some target programs, users have reported that it can take many hours just to compute the distance file. For example, AFLGo spent nearly 2 hours compiling and instrumenting cxxfilt (Binutils) to generate the distance file, which is a non-negligible time cost.

4.3.2 | Similarity

The similarity is a metric that was first proposed by Chen et al. in Hawkeye [82], which measures the similarity between the execution trace of the seed and the target execution trace on the function level. The intuition is that seeds covering more functions in the "expected traces" will have more chances to mutate and reach the targets.

We use the execution tree example in Figure 4 to illustrate how the metric of similarity works. Suppose node T1 is the target basic block, and execution trace A→C→F→L→Q→T1 is the expected trace to target T1. There are two execution traces, A→C→F→K→N and A→C→F→L→P. We can say that trace A→C→F→L→P is more similar to the expected trace A→C→F→L→Q→T1 than trace A→C→F→K→N. This is because trace A→C→F→L→P covers four basic blocks that are overlapped with the expected trace, which is more than trace A→C→F→K→N (only covers three). Thus, trace A→C→F→L→P is regarded as closer to the target than trace A→C→F→K→N.

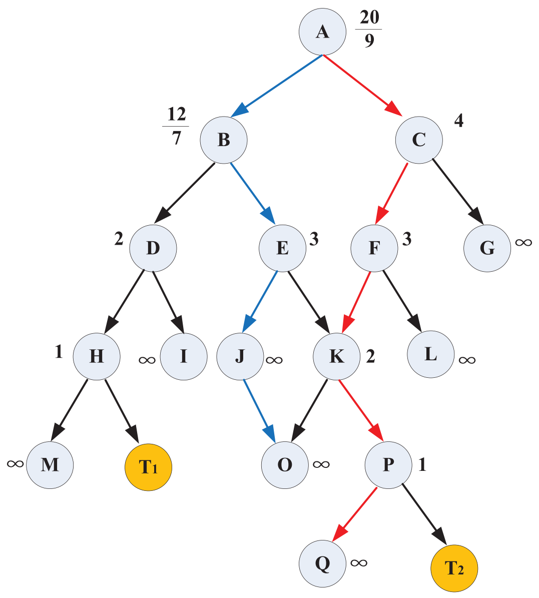


FIGURE 3 Illustration of the distance metric.

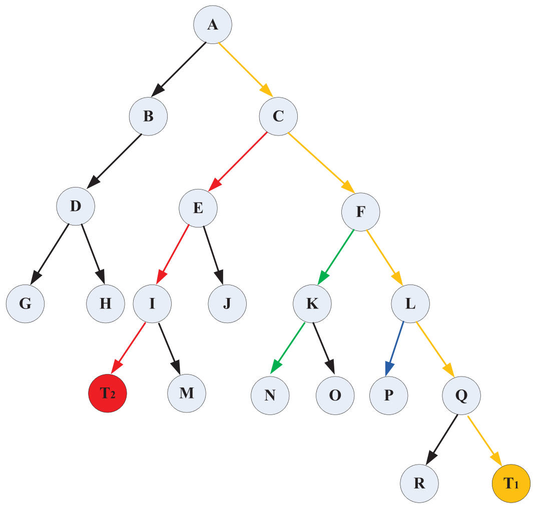


FIGURE 4 Illustration of the similarity metric.

Hawkeye [82] combines the basic block trace distance with covered function similarity for seed prioritization and power scheduling. LOLLY [83] uses a user-specified program statement sequence as the target and takes the seed's ability to cover target sequences (i.e. sequence coverage) as a metric to evaluate the seed. Berry [87] upgraded LOLLY by taking into account the execution context of target sequences. This enhances the target sequences with "necessary nodes" and uses the similarity between the target execution trace and the enhanced target sequence to prioritize the seeds. The similarity is then enriched to cover other target forms, such as operations, bug traces, and labeled locations. Formally, the similarity is *the degree of overlap between the current status and target status of a certain metric, where the metric includes the length of bug traces and the number of covered locations, covered operations, or covered functions.*

UAFL [35] uses operation sequence coverage to guide the test case generation to progressively cover the operation sequences that are likely to trigger use-after-free vulnerabilities. UAFuzz [24] also uses a sequence-aware target similarity metric to measure the similarity between the execution of a seed and the target use-after-free bug trace. SAVIOR [100] prioritizes seeds that have a higher potential to trigger vulnerabilities based on the coverage of labels predicted by UBSan [105]. TortoiseFuzz [27] differentiates edges that are closely related to sensitive memory operations and prioritizes seeds based on the sensitive edge hit count in their execution paths.

For comparison, similarity-based metrics are better able to handle multi-target fitting than distance-based alternatives. This is because the distance-based metric is designed primarily for single-target. When there are multiple targets, a distance-based metric would calculate the distances to different targets one by one, which is less efficient. However, the similarity-based metric can handle multiple targets at one time. For example, in Figure 4, when analyzing execution trace A→ C→ F→ K→ N, we can calculate its similarity (i.e. the number of overlapped basic blocks) with the expected traces to target T1 and T2 simultaneously, which is more efficient. Furthermore, similarity-based metrics can include the relationships between targets, such as the ordering of the targets [24]. Finally, a distance-based metric is measured at the basic block level, which would introduce considerable overheads, while a similarity-based metric can be extracted from a relatively high level to improve overall efficiency.

4.3.3 | Vulnerability prediction models

Researchers also use vulnerability prediction models to quantify how likely a seed can reach a target. Using a deep learning-based model, the vulnerable probability of a function can be predicted, and each basic block in the vulnerable function is given a *Static Vulnerable Score* to measure the vulnerable probability. Then for each input, the sum of the static vulnerable score of all the basic blocks on the execution path is used as a fitness score to prioritize inputs with higher scores [30, 31]. Figure 5 illustrates how the vulnerable score metric works in V-Fuzz [30]. SVS is the static vulnerable score for each basic block predicted by the deep learning-based model. We assume that there are two inputs i_1 and i_2, and the execution paths of the two inputs are $path_1$ and $path_2$, respectively. Suppose $path_1$ is $b_1 \rightarrow b_2 \rightarrow b_4$, and $path_2$ is $b_1 \rightarrow b_3 \rightarrow b_6 \rightarrow b_8$. The fitness score of input i_1 and i_2 are f_1 and f_2, respectively. Then, $f_1 = 2 + 5 + 8 = 15, f_2 = 2 + 1 + 1 + 2 = 6$. As f_1 is larger than f_2, the input i_1 will be selected as a favoured seed.

TAFL [84] extracts semantic metrics of the PUT and uses static semantic analysis to label regions, including sensitive, complex, deep, and rare-to-reach regions, that have a higher probability of containing vulnerabilities and strengthen fuzzing towards such regions. Joffe [9] uses crash likelihood generated by a neural network to direct fuzzing towards executions that are crash-prone. The probability-based metric can combine seed prioritization with target

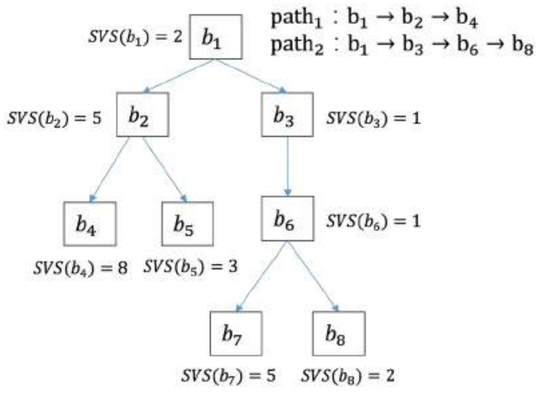


FIGURE 5 Illustration of the vulnerable score metric.

identification to direct fuzzing toward potentially vulnerable locations without relying on the source code. Using deep learning models, a probability-based metric can be extended to targeting properties other than crashes, such as information leaks, exploits, as well as specific crash types, and different resource usages. Besides, deep learning methods have been proven to be able to detect several types of vulnerabilities simultaneously [30]. However, a major weakness is that the accuracy at present is to some extent limited.

4.3.4 | Customized fitness metrics

Apart from the above categories, researchers also propose customized metrics for DGF. Wüstholz et al. [85] used online static look-ahead analysis to determine a path prefix for which all suffix paths cannot reach a target location. Directed fuzzing is then enabled by strategically scheduling the energy of fuzzing to stress the path prefix that might reach the target locations. KCFuzz [96] defines the parent nodes in the path to the target as keypoints and directs fuzzing using keypoint coverage. CAFL [93] aims to satisfy a sequence of constraints (i.e. the combination of a target site and the data conditions) instead of reaching a set of target sites. It defines the distance of constraints as how well a given seed satisfies the constraints and prioritizes the seeds that better satisfy the constraints in order. AFL-HR [81] and HDR-Fuzz [101] adopt a vulnerability-oriented fitness metric called *headroom*, which indicates how closely a test input can expose a hard-to-manifest vulnerability (e.g. buffer or integer overflow) at a given vulnerability location. PERF-FUZZ [37] uses the new maxima of execution counts for all program locations as feedback to generate pathological inputs. To systematically measure fitness, a customized fitness metric also takes into account multiple dimensions simultaneously, including basic code coverage, block weight, number of state transitions, execution time, and anomaly count [15, 86]. In addition, nonfunctional properties such as memory usage [34] and control instability of robotic vehicles [36] can also be used to direct fuzzing.

4.3.5 | Distribution of fitness metrics

Based on the introduction of fitness metrics in the previous subsections, we use Figure 6 to show the distribution of fitness metrics among the tools we investigated. From Figure 6, we can see that the distance metric accounts for the most (36%), followed by the fitness metric of path/sequence coverage, accounting for 21%. Notably, as new fitness metrics, vulnerability prediction models (i.e. vulnerable probability) and customized fitness metrics (i.e., edge hit rate) are important, accounting for 10% and 7%, respectively. Besides, the multi-dimensional fitness metric is also prevalent, accounting for 12%. Thus, we can conclude that the distance metric is still the major fitness metric for DGF, but new fitness metrics are growing fast.

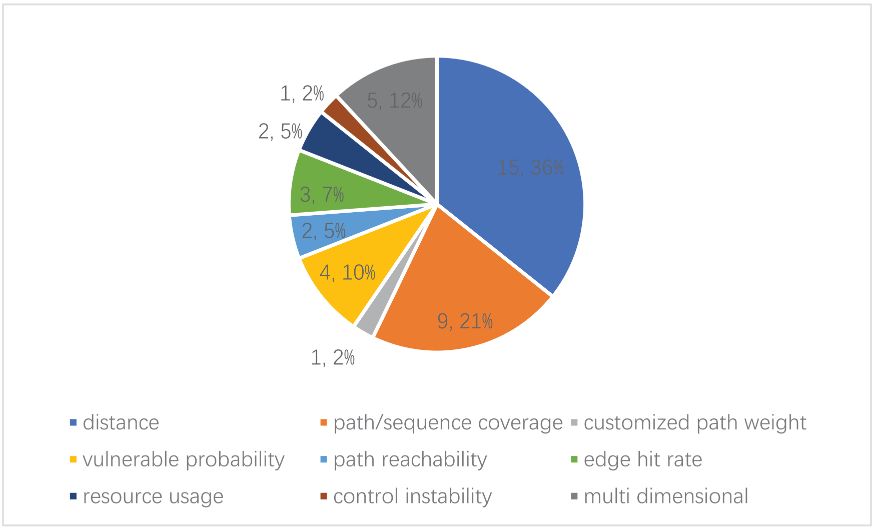


FIGURE 6 Distribution of different fitness metrics among the tools we investigated.

4.4 | Fuzzing optimization

Since a native fuzzer that uses randomly generated test inputs can hardly reach deep targets and is less effective at triggering deep bugs along complex paths, various program analysis techniques, such as static analysis, control-flow analysis, data-flow analysis, machine learning, semantic analysis, and symbolic execution, have been adopted to enhance the directedness of reaching corner cases and flaky bugs. Figure 7 shows the statistics of mainstream optimization techniques used in DGF.

Among the tools investigated, 71% of them relied on the control-flow analysis to evaluate seeds and determine the reachability to the targets; 60% of them leverage static analysis to automatically identify targets [100] and extract information from the PUT [82, 85]; 21% use data-flow analysis (mainly taint analysis) to identify the relationship between the input and the critical program variables [23, 98, 108] or to optimize mutator scheduling [35]; 12% use machine learning to predict vulnerable code [30] and filter out unreachable inputs [91]; 12% integrate symbolic (concolic) execution to solve complex path constraints [23, 28, 87, 100]; and finally, 14% adopt semantic analysis to identify vulnerable targets automatically [22, 28, 84] and learn input fields semantics to optimize mutation. The next section will discuss the key steps of greybox fuzzing and how they are optimized for directedness.

4.4.1 | Input optimization

A good seed input can drive the fuzzing process closer to the target location and improve the performance of the later mutation process. According to Zong et al., on average, over 91.7% of the inputs of AFLGo cannot reach buggy code [91]. There are thus many opportunities to increase the ability of DGF by enhancing the input generation. Dynamic taint analysis [98] and semantic information [22] can assist in generating valid input that matches the input format [89, 108]. These techniques also increase the probability of hitting vulnerable functions [22] or security-sensitive program sites, such as maximizing the likelihood of triggering memory corruption bugs [98]. Except that, FuzzGuard [91] utilizes a deep-learning-based approach to predict and filter out unreachable inputs before exercising them, which saves time that can then be spent on real execution. BEACON [92] prunes infeasible paths (i.e. paths that cannot reach the target code at runtime) with a lightweight static analysis, which can reject over 80% of the paths executed during fuzzing.

4.4.2 | Seed prioritization

The core of DGF is the prioritization of seeds (for mutation) that are closest to the targets. DGF implementation is effectively the act of closest seed-target relation prioritization. No matter what kind of fitness metric it adopts, seed prioritization is mainly realized based on control-flow analysis. Distance-based approaches [21, 23, 24, 33, 82, 88, 89]

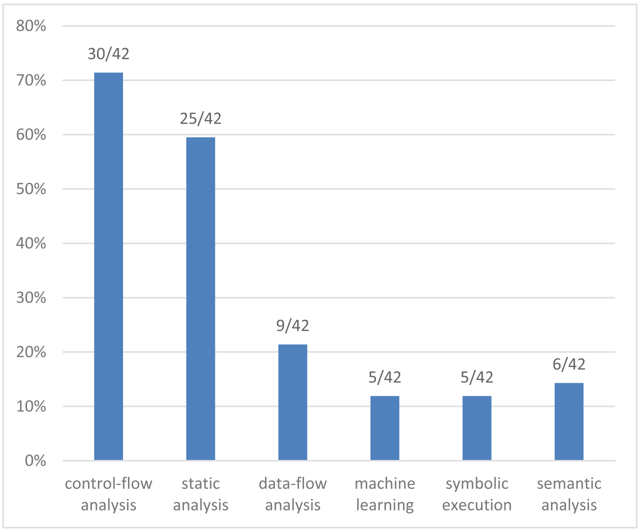


FIGURE 7 Statistics of mainstream optimization techniques used in directed greybox fuzzing (DGF).

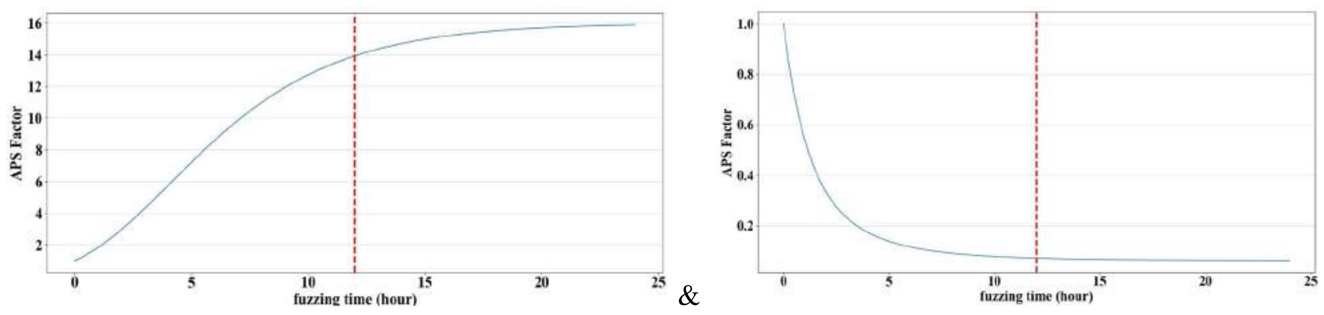
calculate the distance to the target basic blocks from the number of edges in the call and control-flow graphs of the PUT. Similarity-based approaches [24, 27, 35, 83, 87] take the seed's ability to cover the target edges on the control-flow graph as a metric to evaluate the seed. Prediction model-based approaches [30, 31] also rely on the attributed control-flow graph (i.e. using a numerical vector to describe the basic block in a control-flow graph, where each dimension of the vector denotes the value of a specific attribute of the basic block) to represent a binary program and extract features for deep learning. A further point to note is that directed hybrid fuzzing [23, 28, 87, 96, 100] combines the precision of DSE and the scalability of DGF to mitigate their individual weaknesses. DGF can prioritize and schedule input mutation to get closer to the targets rapidly, while DSE can reach more in-depth code by solving complex path constraints.

4.4.3 | Power scheduling

After being selected, the seeds nearest to their targets are subjected to greater fuzzing opportunities by assigning more power; that is, more inputs are produced by mutating them. Whereas AFL uses execution trace characteristics such as trace size, PUT execution speed, and order in the fuzzing queue for power scheduling, most directed greybox fuzzers use simulated annealing to allocate energy. Unlike traditional random walk scheduling, which always accepts better solutions and may be trapped in a local optimum, simulated annealing accepts a solution that is worse than the current one with a certain probability, so it is possible to jump out of local optima and reach the globally optimal solution [83].

To illustrate simulated annealing-based power scheduling in a straight way, we employed AFLGo to test libxml2 and collected the simulated annealing algorithm factor (APS Factor) of seeds with normalized distances of 0.1 and 0.9. As shown in Figure 8, the x-axis represents the fuzzing time ranging from 0 to 24 hours, and the y-axis denotes the variations of the APS Factor. Meanwhile, we set the time threshold for simulated annealing in AFLGo to 12 hours, marked by a red dotted line. Additionally, we adopted an exponential growth mode for energy over time (i.e. exp mode in AFLGo). Figure 8a illustrates the increasing trend of the APS Factor with a normalized distance of 0.1 over time. As fuzzing progresses, the APS Factor with a normalized distance of 0.1 exhibits exponential growth and eventually converges to 16. On the other hand, Figure 8b demonstrates the decreasing trend of the APS Factor with a normalized distance of 0.9 over time. With continuous fuzzing, the APS Factor with a normalized distance of 0.9 shows exponential decay and eventually converges to 1/16.

AFLGo [21] was the first to use a simulated annealing-based power schedule to gradually assign more energy to seeds that are closer to the target locations while reducing energy for distant seeds. Hawkeye [82] added prioritization to simulated annealing to allow seeds that are closer to the target to mutate first. AFLChurn [26] proposes a byte-level power scheduling based on ant colony optimization which assigns more energy to bytes that generate more “interesting” inputs. LOLLY [83] and Berry [83] optimized simulated annealing-based power schedules with a temperature threshold to coordinate the cooling schedule in both the exploration and exploitation stages. In the exploration stage, the cooling schedule randomly mutates the provided seeds to generate many new inputs, while in the exploitation stage, it generates more new inputs from seeds that have higher sequence coverage, which is similar to the traditional gradient descent algorithm [83]. In addition to simulated annealing, GREYHOUND [15] also adopts a custom generational particle swarm algorithm, which is better suited for the nonlinear and stochastic behavior of the protocol model.



(a) The variations of APS Factor for seeds with a normalized distance of 0.1.

(b) The variations of APS Factor for seeds with a normalized distance of 0.9.

FIGURE 8 (a, b) Illustration of the simulated-annealing-based power scheduling.

4.4.4 | Mutator scheduling

Optimizing mutator scheduling is another viable way of bettering directed fuzzing. Reasonable scheduling of mutators can enhance the directedness of inputs by improving the precision and speed of seed mutation. A viable approach is to first classify mutators into different granularities, such as coarse-grained and fine-grained [22, 30, 82, 84], and then dynamically adjust them according to the actual fuzzing states. Coarse-grained mutators are used to change bulk bytes during mutations to move the execution towards the "vulnerable functions", while fine-grained only involve a few byte-level modifications, insertions, or deletions, so as to monitor the "critical variables" [22]. The fuzzer gives a lower chance of coarse-grained mutation when a seed can reach the target function. Once the seed reaches targets, the time for fine-grained mutations increases as coarse-grained mutations decrease. In practice, the scheduling of mutators is controlled by empirical values [30, 82]. Situ et al. [84] give two empirical observations—that (1) coarse-grained mutators outperform fine-grained mutators on path growth and (2) the use of multiple mutations offers improved performance compared to each individual mutation.

4.4.5 | Mutation operations

Data-flow analysis, such as taint analysis, can reflect the effect of the mutation in the generated inputs; thus, it is helpful to optimize both mutation operations and input generation. RDFuzz [88] leverages a disturb-and-check method to identify and protect "distance-sensitive content" from the input, that is, the critical content to maintain the distance between the input and the target, and once altered, the distance becomes larger. Protecting such content during mutation can help to approach the target code location more efficiently. UAFL [35] adopts information flow analysis to identify the relationship between the input and the program variables in the conditional statement. It regards input bytes that are more likely to change the values of the target statement with higher "information flow strength" and assigns higher mutation possibility for them. The higher the information flow strength, the stronger this byte influences the values of the variables. SemFuzz [22] tracks the kernel function parameters that the critical variables depend on via backward data-flow analysis. TIFF [98] infers input type by type-based mutation to increase the probability of triggering memory corruption vulnerabilities. It leverages in-memory data-structure identification to identify the types of each memory address used by the application and uses dynamic taint analysis to map what input bytes end up in what memory locations. Nevertheless, data-flow analysis usually enlarges the run-time overhead.

4.5 | Base tools

Based on the statistics in Table 1, we use Figure 9 to show the distribution of the base tools. As Figure 9 shows, 62% (20/42) of the DGF tools are built on AFL, which is similar to the situation of CGF. As AFL is a well-structured framework with good performance, it is suitable for further development, including DGF research. Most DGF tools

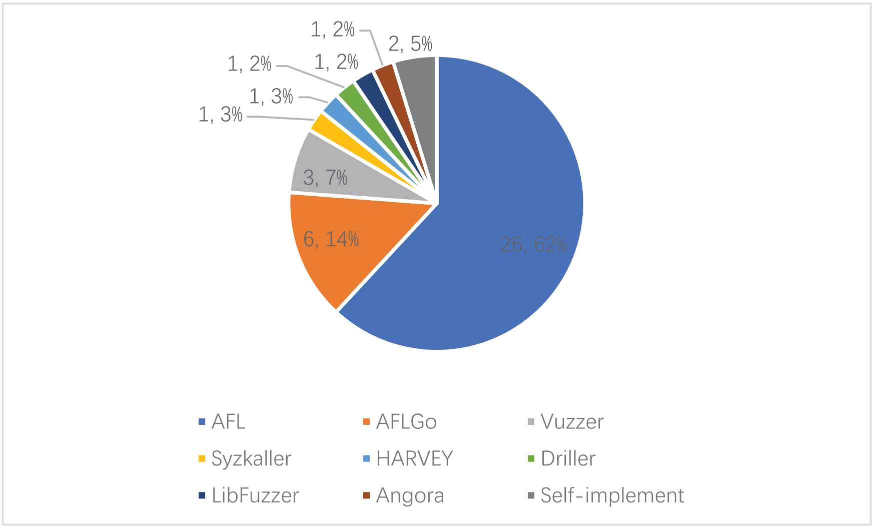


FIGURE 9 Distribution of the base tools.

with customized fitness metrics would adopt AFL as the base tool. The second most used (6/42) base tool is AFLGo. Since AFLGo is the first DGF tool based on the basic block distance, it is suitable for tools with distance-based fitness metrics. In addition, some base tools are adopted owing to the test requirement. For example, Syzkaller is used as the base tool for DGF in the kernel, VUzzer is used as the base tool for DGF at the binary level, and HARVEY is used as the base tool for testing Ethereum smart contracts.

There are also tools built without a based tool, such as TOFU and RVFuzzer. For TOFU, though it uses a distance-based fitness metric, the author did not mention any base tools in their paper but stated that "the high-level structure of TOFU is similar to that of AFL". Since TOFU is not open-sourced, we can only infer that TOFU is self-devised. The reason may be that it relies on the compile tool WLLVM to extract ICFG, and WLLVM is incompatible with the commonly used base tools. As for the RVFuzzer, since it is used to test robotic vehicles, most of the existing base tools, such as AFL and AFLGo, cannot be used directly due to the source code requirement and environmental requirements. In such a situation, the author would develop a customized tool, which is more usable, more lightweight, and more efficient for the target system.

The main advantage of using base tools is the convenience of implementing tools. The already widely used base tool can provide a reliable platform to build new tools. For example, when developing a distance-based DGF tool based on AFL, we only need to modify a few modules (e.g. the fitness metric) in the framework, which is very time-saving. However, the major disadvantage lies in the compatibility. When the testing target has restrictions, such as only having binary code or has different running environment, the commonly used base tools are not suitable or less efficient. A self-devised customized tool is needed, which is more usable, more lightweight, and more efficient for the target environment. For example, RVFuzzer for testing robotic vehicles.

5 | CHALLENGES FACED BY DGF

5.1 | Performance deduction

To realize directedness in fuzzing, most researchers use additional instrumentation and data analysis, for example, static analysis, symbolic execution, taint analysis, and machine learning. However, such additional analysis inevitably incurs performance deduction. The analysis cost is one of the biggest drawbacks of almost all the directed fuzzing techniques. To evaluate the performance of directed greybox fuzzers, researchers usually focus on the ability to reach targets or expose bugs, using metrics such as *Time-to-Reach* (the length of the fuzzing campaign until the first testcase that reaches a given location) or *Time-to-Exposure* (the length of the fuzzing campaign until the first testcase that exposes a given error [21]) to measure the performance of directed greybox fuzzers, while ignoring the measurement of performance overhead. However, for a given fuzzing time budget, higher efficiency means more fuzzing executions and consequently, more chance to reach the target. Thus, optimizing efficiency is a major challenge to improve directedness. Based on our investigation, we summarize the following solution to improve DGF efficiency.

- **Move the heavy execution-independent computation from run-time to compile-time.** For example, AFLGo measures the distance between each basic block and a target location by parsing the call graph and intra-procedural control flow graph of the PUT. Since both parsing graphs and calculating distances are very time-consuming, AFLGo moves most of the graph parsing and distance calculation to the instrumentation phase at compile-time in exchange for efficiency at run-time. Such compile-time overhead can be saved when a PUT is tested repeatedly.
- **Filter out the unreachable inputs to the target before execution.** For example, FuzzGuard [91] utilizes a deep-learning-based approach and BEACON [92] uses a lightweight static analysis to find infeasible paths to the targets in advance, which can save over 80% of the path execution during fuzzing;
- **Use more light-weight algorithms.** For example, AFLChurn [26] leverages lightweight ant colony optimization instead of expensive taint analysis to find "interesting bytes" and realize a byte-level power scheduling;
- **Leverage parallel computing.** For example, HDR-Fuzz [101] uses another core to run AddressSanitizer in parallel and provides guidance to the directedness. Large-scale parallel fuzzing [109, 110] can also be adopted to improve efficiency further.

However, these approaches might be less effective for certain situations. For example, for the continuously evolving software, moving analysis from run-time to compile-time may not get obvious benefit as it will have to be rerun after every single code change during compile time. Filtering out unreachable inputs and pruning infeasible paths inevitably introduce false reports, making unreachable inputs and infeasible paths remain and such computational-complex paths and inputs would influence the performance.

In recent years, two state-of-the-art techniques, namely, BEACON [92] and SelectFuzz [111], have significantly improved the speed of bug exposure by reducing the cost of fuzzing. BEACON leverages symbolic execution to analyze the feasibility of different paths and eliminates those that cannot lead to the target, thereby reducing the overall fuzzing cost. In a similar vein, SelectFuzz conducts a preliminary analysis of the reachability of basic blocks and selectively instruments and calculates seed distances for blocks that are reachable. As a result, the overhead associated with instrumentation and seed distance calculation is minimized. By this means, SelectFuzz [111] avoids exploring irrelevant code, further reducing the cost of fuzzing. By adopting these strategies, both BEACON [92] and SelectFuzz [111] effectively decrease the fuzzing cost by minimizing the exploration of code that does not contribute to reaching the desired targets, thereby enhancing the speed of bug exposure. In contrast, other techniques such as Windranger [97], CAFL [93], and FuzzGuard [91] actually increase the cost of fuzzing due to their requirements for analyzing and collecting DBBs and path constraints, or collecting and filtering seeds. However, in our evaluation, we found that the additional fuzzing cost incurred by these techniques does not have a significant impact on the overall fuzzing throughput. Furthermore, the fitness metrics and fuzzing strategies proposed by these techniques can effectively guide DGF to reach targets faster. Considering the improved speed of bug exposure achieved by these fitness metrics, the additional cost of fuzzing is deemed acceptable. This has achieved a tradeoff between fuzzing cost and bug exposure.

5.2 | Equal-weighted metrics bias seed prioritization

In most of the state-of-the-art directed greybox fuzzers, the seed prioritization is based on equal-weighted metrics, that is, treat each branch jump in the control-flow graph as having equal probability. We take the widely used distance-based metric as an example, where the distance is represented by a number of edges, namely, the transitions among basic blocks. However, such measurement ignores the fact that different branch jumps have different probabilities to take, and thus, biases the performance of directed fuzzing.

Figure 10 shows a control-flow graph fragment of a simple example to illustrate the problem. Suppose input x is an integer ranging from 0 to 9. Obviously, the probability of jumping from node A to node C is 0.1, and from node A to node B is 0.9. We can also compute the probabilities of other jumps by the branch conditions. When using a distance-based metric, the distance of A $\rightarrow$ C is shorter than that of A $\rightarrow$ G because A $\rightarrow$ C has only one jump but A $\rightarrow$ G has three jumps. However, when taking the jump probability into account, the probability of A $\rightarrow$ C is 0.1, while the probability of A $\rightarrow$ G is $0.9 \times 0.7 \times 0.5 \approx 0.3$, which is more likely to be taken than A $\rightarrow$ C and should be considered as "shorter". Thus, it is reasonable to also consider the weight difference when designing the fitness metric. Though this is a hypothetical example, such a problem is realistic and frequent in the real-world program. One common case is when A $\rightarrow$ C represents an execution path through the error-handling code. The error-handling code is usually short and simple, and is used to retrieve resources, such as freeing the allocated memory. Thus, the execution path through the error-handling code to the target is usually short in distance (e.g. one jump). However, since error-handling code is rarely executed, such an execution path has a low probability. If we only consider distance, the path through the error-handling code would be overemphasized, and we would ignore the bug-prone regular code, leading to a bias.

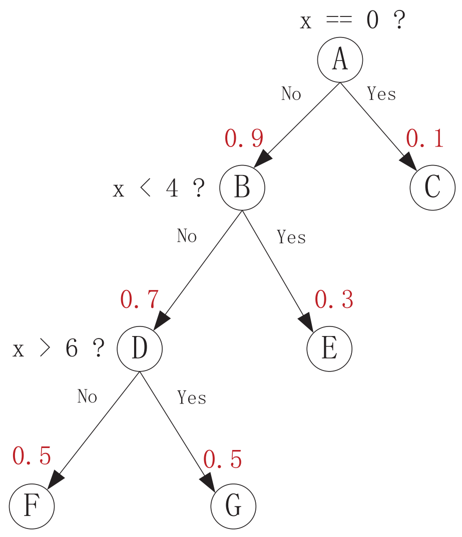


FIGURE 10 Equal-weighted metric incurs bias in distance-based seed prioritization.

One solution is taking branch jump probability into account to construct weighted fitness metrics. In that case, each seed is prioritized by the probability of converting the current execution path to a target path that goes through the target. Since an execution path can be viewed as a Markov chain of successive branches [44], the path probability can be calculated by gathering the probabilities of all the branches within the path. Then the branch probability can be statistically estimated by calculating the ratio based on the Monte Carlo method [27]. By its very nature, the randomness and high throughput of the fuzzing process fulfill the requirements for random and large sampling with the Monte Carlo method. Thus, the distribution density can formally estimate the branch jump probability in a lightweight fashion.

One possible drawback of evaluating the reachability of the target based on probability is the potential run-time overhead. Both the statistical jump counting and the probability calculation introduce extra computation. One way to alleviate performance deduction is interval sampling. Compressing the volume of jump statistics by appropriate sampling can accelerate the probability calculation and alleviate the space requirement for storage. Another way is to accelerate how the meta-data of jump statistics is stored and accessed. On the one hand, the probability-based approach updates the jump statistics very often, and the reachability judgment also requires quick edge tracing. On the other hand, since a jump usually has only two branches, the data distribution (e.g. based on a matrix) would be relatively sparse, which dramatically increases space consumption. Thus, a customized data structure that balances the time and space complexities is required.

5.3 | The global optimum discrepancy in the distance-based metric

When measuring multiple targets with a distance-based metric, one strategy is to seek the global shortest distance between the execution path and all the targets using Dijkstra's algorithm [21, 82, 85, 88, 89]. However, such global optimum might miss local optimal seeds that are closest to a specific target, leading to a discrepancy. In order to elucidate this case, an example is depicted in Figure 11. In this control-flow graph fragment, nodes K and O are the target nodes. For the three seeds under test, one exercises path A→B→D→G→K, one exercises path A→C→E→I→M→N→O, and the last exercises path A→C→E→H→L. Based on the distance formula defined by Böhme et al. [21], the harmonic distances were calculated between each node in the three paths to the two targets—these are labeled in the figure. The global distances for each of the three seeds are $d_{ABDGK} = (4/3+3+2+1+0)/5 \approx 1.47$, $d_{ACEIMNO} = (4/3+3/4+2+3+2+1+0)/7 \approx 1.44$ and $d_{ACEHL} = (4/3+3/4+2+1)/4 \approx 1.27$. Since d_{ACEHL} is the smallest of the three, one should prioritize the seed for path A→C→E→H→L. However, this is unreasonable because path

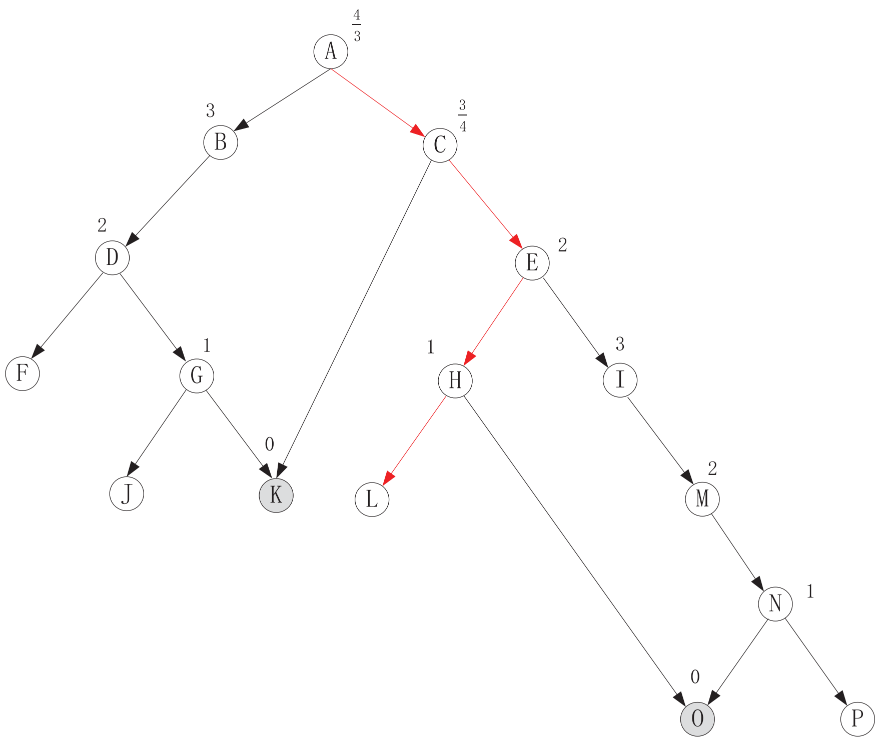


FIGURE 11 Discrepancy introduced by distance-based seed prioritization metric.

A→ B→ D→ G→ K goes through target node K, while path A→ C→ E→ I→ M→ N→ O goes through target O, but path A→ C→ E→ H→ L does not reach any targets. Intuitively, as path A→ C→ E→ H→ L is far from the targets, it should not be prioritized. The efficacy of directed fuzzing is affected when there is more than a single target, as finding the global shortest distance has a discrepancy.

The reason behind such discrepancy is that the distance-based seed measurement only focuses on the shortest path. When there are multiple paths reaching the same target, the longer ones might be ignored, causing a discrepancy in the result. In Figure 11, if the paths A→ C→ K and A→ C→ E→ H→ O are considered, then $d_{ACK} = (4/3 + 3/4 + 0)/3 \approx 0.69, d_{ACEHO} = (4/3 + 3/4 + 2 + 1 + 0)/5 \approx 1.02$. As expected, $d_{ACK} < d_{ACEHO} < d_{ACEIMNO}$. This is because path A→ C→ K and path A→ C→ E→ H→ O are the shortest paths from A to targets K and O, respectively. The shortest path is always prioritized. Such discrepancy is realistic and frequently occurs when three conditions are all met: (1) multiple targets are measured by distance, (2) at least one target has more than one viable path and (3) a seed exercises the longer path and is measured by this distance. Multi-target testing is a frequently used scenario when applying DGF. For example, testing patches by setting code changes as targets. Thus, condition (1) is easy to meet. For condition (2), we also use the error handling code as an example. The error-handling code can be the destination of many functional modules, which means a target in the error-handling code is usually reachable via many paths; thus, condition (2) is also easy to meet. Finally, the satisfaction of condition (3) is uncertain as we cannot guarantee that the longer path is exercised. Only when a seed exercises the longer path, it would be measured by this distance, and a discrepancy occurs.

To avoid such discrepancy, all potential paths to the targets must be accounted for. For example, under a different context, the distances from the calling function to the immediately called function may not be exactly the same. To solve this problem, Hawkeye uses “adjacent-function distance augmentation” based on a lightweight static analysis [82], which considers the patterns of the (immediate) call relation based on the generated call graph to augment the distance that is defined by immediate calling relation between caller and callee. Another strategy for coordinating multi-targets is separating the targets. For each seed, only the minimum distance for all targets is selected as the seed distance, and the seeds are prioritized based on this min-distance [23]. The effect of this is to negate the possibility of biassing into global optimal solutions but at the cost of increasing the time required to hit a given target.

5.4 | Inflexible coordination of the exploration phase and exploitation phase

Another challenge of DGF lies in coordinating the exploration-exploitation trade-off. On the one hand, more exploration is necessary to provide adequate information for exploitation; on the other hand, an overfull exploration would consume many resources and delay the exploitation. It is difficult to determine the boundary between the exploration phase and the exploitation phase to achieve the best performance.

Most directed greybox fuzzers, such as AFLGo, adopt a fixed splitting of the exploration phase and the exploitation phase. The time budgets are preset in the test configuration before testing. Such a scheme is preliminary because the separation point is inflexible and relies on the human experience. Since each PUT is different, such fixed splitting is less adaptive. Once the exploration phase gives way to the exploitation phase, there is no going back even if the direction performance is poor due to insufficient paths.

The efficacy of DGF is determined by how the resources for exploration and exploitation are divided. To elucidate this with a case study, AFLGo was applied to `libxml` using the “-z” parameter of AFLGo to set different time budgets for the exploration phase and compare the performance. As Figure 12 shows, the horizontal coordinate shows the time duration of the test and the vertical coordinate means the minimum distance of all the generated inputs to the target

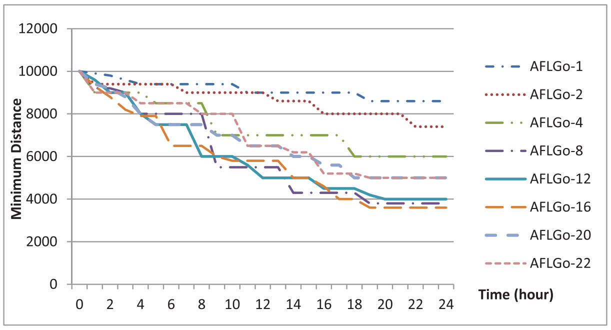


FIGURE 12 Comparison of different splittings of the exploration phase and the exploitation phase.

code areas (min-distance). A small "min-distance" indicates a better-directed performance. The experiments last for 24 hours, and AFLGo-1 means 1 hour of exploration with 23 hours of exploitation, and the rest are similar. From the results, it can be concluded that the splitting of the exploration and exploitation phases affects the performance of DGF and that the best performance (AFLGo-16) requires adequate time for both of the two phases. However, it is difficult to get optimum splitting.

Among the directed fuzzers investigated, only one work tries to optimize the coordination of exploration–exploitation. RDFuzz [88] combines distance and frequency to evaluate the inputs. Low-frequency inputs are required in the exploration phase to improve the coverage, while short-distance inputs are favored in the exploitation phase to achieve the target code areas. Finally, an intertwined testing schedule is used to conduct the exploration and exploitation alternately. However, the classification of the four input types (short/long distance and low/high frequency) is preliminary, and the performance heavily depends on the empirical threshold values.

5.5 | Dependence on the PUT source code

Most of the known DGF works [21, 82, 88] are implemented on top of AFL and inherit AFL's compile-time instrumentation scheme to feedback execution status or calculate distance-based metrics. A significant drawback of such a scheme is the dependence on the PUT source code. Thus, it is unsuitable for testing scenarios where the source code is unavailable, such as commercial off-the-shelf (COTS) software or security-critical programs that rely partly on third-party libraries.

There are multiple reasons that hinder the application of DGF at the binary level. First, **heavy run-time overhead**. A straightforward solution to binary-level testing is through a full-system emulator, such as QEMU [24]. However, emulator-based tools are usually less efficient. For example, the execution speed of vanilla AFL is 2-5 times faster than its QEMU mode [112]. Second, **difficulty in collecting target information**. An open-source PUT can be used to obtain target information from various channels, such as the CVE vulnerability descriptions [27, 28], changes made in the git commit logs [22], and human experience on critical sites in the source code. However, for binary code, we can only extract target information from bug traces [24]. Third, **difficulty in labeling the targets**. For the source code instrumentation approach, the targets can be labeled based on the source code (e.g. cxxfilt.c, line 100). However, it is much more difficult for the binary. Since the binary code is hard to read, it must be disassembled using tools such as IDA Pro [24], and the targets labeled with virtual addresses, which is both inconvenient and time-consuming.

A viable solution to alleviate the performance limitation is hardware assistance, such as Intel Processor Trace (PT). Intel PT is a lightweight hardware feature in Intel processors. It can trace program execution on the fly with negligible overhead (averagely 4.3x faster than QEMU-AFL [113]), which replaces the need for dynamic instrumentation. Using the packet trace captured by Intel PT along with the corresponding binary of the PUT, the execution path of the PUT could be fully reconstructed. There have been attempts of fuzzing with PT [10, 112–114], but it has never been used to DGF yet. For the problem of target identification and labeling at the binary code level, a machine-learning-based approach [30, 31] and a heuristic binary diffing approach [100] can be leveraged to automatically identify the vulnerable code.

6 | APPLICATION OF DGF

DGF has a good application prospect. When a practitioner chooses a directed greybox fuzzer, the first thing to consider is the application scenario. We summarize the following typical scenario for the DGF application.**Patch testing**. DGF can test whether a patch is complete and compatible. A patch is incomplete when a bug can be triggered by multiple inputs [115], but the patch only fixes a part of them. For example, CVE-2017-15939 is caused by an incomplete fix for CVE-2017-15023 [82]. Meanwhile, a patch can introduce new bugs [116]. For example, CVE-2016-5728 is introduced by a careless code update. Thus, directed fuzzing towards problematic changes or patches has a higher chance of exposing bugs. For example, DeltaFuzz [25] and AFLChurn [26] are designed for regression testing. SemFuzz [22] uses code changes from git commit logs. UAFuzz [24] and 1dvul [23] use binary-level comparison to identify patch-related target branches, which are particularly suitable for this scenario.

Bug reproduction. DGF can reproduce a known bug when the buggy input is unavailable. For example, due to concerns such as privacy, some applications (e.g. video players) are not allowed to send the input file. With DGF, the in-house test team can use DGF to reproduce the crash with the method calls in stack trace and some environmental parameters [21]. DGF is also helpful when generating Proof-of-Concept (PoC) inputs of disclosed vulnerabilities with given bug report information [22, 23]. In fact, DGF is in demand because 45.1% of the usual bug reports cannot be reproduced due to missing information and user privacy violations [117]. TortoiseFuzz [27] and DrillerGo [28] utilize CVE vulnerability descriptions as target information, while UAFuzz [24] extracts target information from bug traces, both of which are suitable for this scenario.

Knowledge integration. DGF can boost program testing by integrating the knowledge from a human analyst. Human-in-the-loop can help to overcome roadblocks and explore the program's state space more thoroughly. For example, IJON [29] uses human experience to identify the security-sensitive program sites (e.g. call site of `malloc()` and `strcpy()`) to guide fuzzing towards error-prone parts [29], which are suitable for this scenario.

Result validation. DGF can validate the result of other software testing approaches. Testing approaches such as static analysis and machine learning can help to identify potentially vulnerable targets, though the results are inaccurate. DGF can be used to refine the results by removing false positives. Tools like V-Fuzz [30], SUZZER [31], DeFuzz [32] and ParmeSan [33] are suitable for this scenario.

Energy saving. Another interesting application of DGF is when the testing resource is limited. For example, IoT devices fuzzing. Under this circumstance, identifying critical code areas to guide testing is more efficient than testing the whole program in an undirected manner, which can save time and computational resources being spent on nonbuggy code regions. GREYHOUND [15] and RVFUZZER [36] are designed for Wi-Fi clients and robotic vehicles respectively, and are both suitable for this scenario.

Non-crash bug detection. Finally, DGF can detect non-crash bugs based on customized indicators. For example, it can find uncontrolled memory consumption bugs under the guidance of memory usage [34] and find algorithmic complexity vulnerabilities under the guidance of resource usage [38, 102].

The second thing to consider is the test conditions. Of these, the source code availability is of vital importance. In order to realize directed fuzzing, researchers use additional instrumentation and data analysis in the fuzzing process. Taking AFLGo as an example, when instrumenting the source code at compile-time, the control-flow graphs and call graphs are constructed via LLVM's link-time-optimization pass. After this, AFLGo measures the distance between each basic block and a target location by parsing the call graph and intra-procedural control-flow graph of the PUT. For the tools reviewed herein, 81% rely on the PUT source code.

Since both parsing graphs and calculating distances are very time-consuming, preprocessing is required. AFLGo moves most of the program analysis to the instrumentation phase at compile-time in exchange for efficiency at run-time. Notwithstanding this, AFLGo still spent nearly 2 hours compiling and instrumenting `cxxfilt` (Binutils) [24], which is a non-negligible time cost. For cases where the source code is unavailable, there are three challenges to consider—the heavy run-time overhead caused by QEMU [24], the difficulty in collecting target information, and the difficulty in labeling targets, all of which result in inconvenience and reduced efficiency (this is discussed in detail in Section 5.5).

Last but not least, the number of targets and the number of testing objectives also affect the choice of a tool. When there are multiple targets, the relationship among targets is also exploitable. For example, UAFL [35] takes into account the operation ordering of target sequences to find complex behavioral use-after-free vulnerabilities (will discuss in Section 9.1). Most of the tools investigated tend to only focus on optimizing a single objective, such as covering specific targets. A multi-objective optimization is a practical option that meets the demand of optimizing more than one fitness metric simultaneously. For example, Memlock [34] generates test sets that maximize memory usage and code coverage at the same time (will be discussed in detail in Section 9.3).

7 | THREATS TO VALIDITY

First, our search method in this survey mainly focuses on the top venue works, which might miss some works that are related to DGF but not published in top venues. Second, since this survey was finished by 2022.5, it only collected papers published from 2017.1 to 2022.5; thus, it might miss some new works that were published after 2022.5. These new works might bring new techniques that can address the challenges proposed in this paper. Nevertheless, this paper can still reflect the main research progress and future trends.

8 | RELATED WORKS

Fuzzing surveys. So far there have been several surveys on fuzzing. As far as we can find, Fell [118] conducted the first review on fuzzing in 2017, which introduced the basic scheme of fuzzing and the existing tools on protocols fuzzing and web browser fuzzing. Li et al. presented an overview of fuzzing solutions by the year 2017 and discussed techniques that could make the fuzzing process smarter and more efficient, including static analysis, taint analysis, machine learning, and format methods. Liang et al. [6] summarized 18 typical fuzzers ranging from 2004 to 2017. They discussed the key obstacles and some state-of-the-art technologies that aim to overcome or mitigate these obstacles. In 2019, Manès et al. [2] gave a detailed survey on 63 modern fuzzers. They explored the design decisions at every stage of fuzzing by surveying the related literature. Böhme et al. [119] summarized the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting. Finally, Zhu et al. [119] gave the most up-to-date survey on fuzzing to narrow down the gaps between the entire input space and the defect space. The survey reviews and analyses the gaps as well as their solutions, considering both breadth and depth. In addition to surveys on common fuzzing, there are also surveys focused on subclasses of fuzzing. Eisele et al. reviewed the fuzzing approaches for embedded systems [120]. They gave a formal definition of embedded fuzzing and grouped the approaches according to how the execution environment is served to the system under test. Saavedra et al. [121] reviewed the machine learning applications in fuzzing, including deep learning, neural networks, and reinforcement learning. They discussed successful applications of machine learning to fuzzing, such as input generation, seed selection, and corpus minimization. Wang et al. also gave a review of fuzzing based on machine learning techniques [122]. They identified six different stages in which machine learning has been used and studied the machine learning-based fuzzing models from the selection of machine learning algorithm, preprocessing methods, datasets, evaluation metrics, and hyperparameters setting. They also assessed the performance of the machine learning models based on the frequently used evaluation metrics. Zhang et al. also gave a preliminary survey on directed fuzzy technology [123]. However, this survey only gave a brief introduction and lacked a detailed comparison of different approaches.

Generational fuzzing. Generating syntactically and semantically valid inputs can improve the efficiency of fuzzers in code coverage and bug detection [124]. To achieve this goal, researchers have proposed generation-based fuzzing, which utilizes grammar or models to describe the input structure and generate syntactically correct inputs [124–128]. This approach has been widely employed in fuzzing targets that require highly structured inputs, such as parsers, protocols, and compilers [124, 125, 127–130]. Utilizing a well-defined grammar or model can significantly enhance the fuzzer. However, creating a manual grammar requires substantial effort [131–133]. To alleviate this burden, some researchers have explored learning the grammar from existing test cases using machine learning techniques [134–136]. Nevertheless, as pointed out in BeDivFuzz [124], having only syntactically correct inputs may not be sufficient to explore deeper regions of programs. Recent studies have combined mutation-based fuzzing with generation-based fuzzing and incorporated coverage mechanisms in greybox fuzzing to improve the efficiency of fuzzers. Zest [126] integrates coverage feedback to generate inputs with high semantic coverage. BeDivFuzz [124] follows a similar mechanism but extends it with structural mutation strategies. DGF can also be enhanced by incorporating generation-based techniques to generate highly structured test cases that satisfy more constraints and reach the intended targets.

Anti-fuzzing. Whitehouse et al. [137] introduced the concept of anti-fuzzing and proposed strategies such as fake crashes and performance degradation. David et al. [138] identified four attack vectors against fuzzers, including execution speed, crash masking, fuzzer detection, and feedback mechanism detection. FUZZIFICATION [139] and ANTI-FUZZ [140] subsequently proposed countermeasures for degrading fuzzers. For example, they injected a large number of fake bugs into the target application to attack the feedback mechanism in CGF, transformed explicit data flows into implicit data flows for anti-hybrid fuzzing, and inserted delay codes in cold paths to slow down the fuzzer. Some of these countermeasures are also employed by VALL-NUT [141]. However, certain strategies, such as executing delay codes, are only applied when the inputs trigger paths that regular users rarely reach but fuzzers are prone to fall into. In

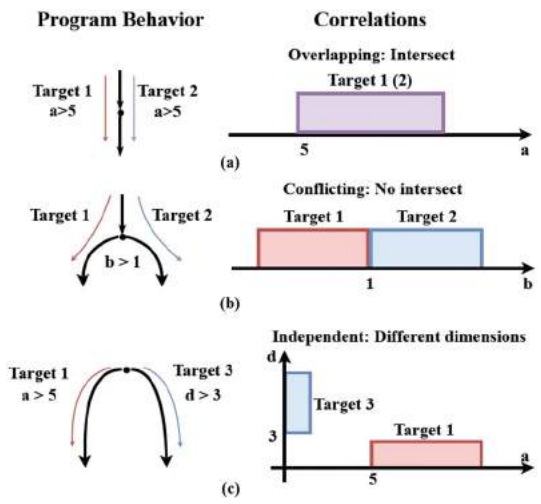


FIGURE 13 Correlations among different targets.

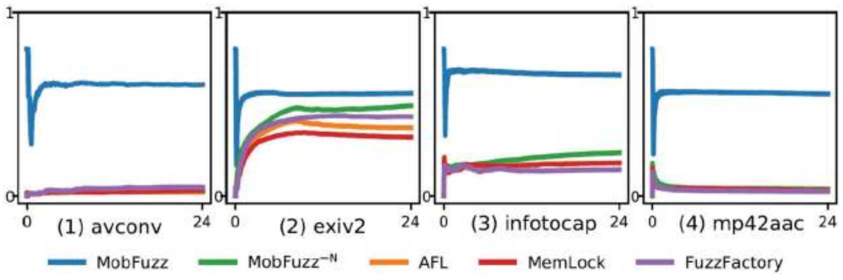


FIGURE 14 Multi-objective optimization performance on good seeds.

the case of DGF, since the scheduling process of target-DGF is guided by predefined targets, a target that is not present in those paths may result in the fuzzer generating fewer test cases that cover the cold paths. As for behaviour-DGF, such as memlock, memory consumption is the fitness metric, which is barely affected by anti-fuzzing techniques. This may mitigate the effectiveness of these path-based strategies in antifuzzing, and these passive antifuzzing strategies may have a lesser impact on DGF compared with CGF. Nevertheless, No-Fuzz [142] proposed strategies for accurately detecting binary-only instrumentations used by fuzzers, such as timing-related techniques and execution frequency examination. Once fuzzers are detected, mitigation techniques are implemented. Jiang et al. [143] utilized inconsistent instructions to detect the emulation technique used in binary-only fuzzing. These techniques have been proven to have strong detection capabilities for fuzzers and antifuzzing, including DGF.

Fuzzing cost. BEACON [92] leverages symbolic execution to analyze the feasibility of different paths and eliminates those that cannot lead to the target, thereby reducing the overall fuzzing cost. In a similar vein, SelectFuzz [111] conducts a preliminary analysis of the reachability of basic blocks and selectively instruments and calculates seed distances for blocks that are reachable. As a result, the overhead associated with instrumentation and seed distance calculation is minimized. In contrast, other techniques such as Windranger, CAFL, and FuzzGuard actually increase the cost of fuzzing due to their requirements for analyzing and collecting DBBs and path constraints or collecting and filtering seeds. However, since the fitness metrics and fuzzing strategies proposed by these techniques can effectively guide DGF to reach targets faster, considering the improved speed of bug exposure achieved by these fitness metrics, the additional cost of fuzzing is deemed acceptable, which can still achieve a tradeoff between fuzzing cost and bug exposure.

9 | CONCLUSION AND PERSPECTIVES ON FUTURE TRENDS

DGF is a practical and scalable approach to software testing, which can be applied to specific scenarios, such as patch testing, bug reproduction, and special bug detection. Modern DGF has evolved from reaching target locations to detecting complex deep behavioral bugs. This paper conducts the first in-depth study of DGF based on the review of 42 state-of-the-art tools related to DGF. After summarizing the recent progress in DGF and the challenges faced by DGF, we make the following suggestions in terms of the perspectives and future trends of DGF, aiming to facilitate and boost research in this field.

9.1 | Exploitation of the relationship between targets

When there are multiple targets in a targeted fuzzing task, how to coordinate these targets is another challenge. Although 86% (36/42) of the fuzzers we investigated support multi-targets, only four of them paid attention to the relationship among targets. For multiple targets to be reached, exploiting the relationship among targets is meaningful for optimizing DGF. If the targets are unrelated, weights can be assigned to them to differentiate the importance or probability. Alternatively, the hidden relationship can be extracted and exploited to improve directedness. For example, UAFL [35] takes into account the operation sequence ordering when leveraging target sequences to find use-after-free vulnerabilities. This is because, to trigger such behavioral complex vulnerabilities, one needs not only to cover individual edges but also to traverse some longer sequences of edges in a particular order. Such a method can be extended to detect semantic bugs, such as double-free and API misuse. Berry [87] enhanced the target sequences with execution context (i.e. necessary nodes required to reach the nodes in the target sequences) for all paths. Similarly, KCFuzz [96] regards the parent nodes in the path to the target as keypoints to cover. CAFL [93] regards the data conditions along the path to the target as constraints and drives the seeds to satisfy the constraints in order to finally reach the target.

Listing 1: A running example for multitargets.

```
01  int foo(){
02    signed a,b,c,d = parse_input();
03    int flag;
04
05    if(a>5){
06      if(b>1){
07        flag=0;
08      }
09      else {
10        flag=1;
11        if(...) {...} //target-irrelevant
12      }
13      ...
14      if(flag){
15        target1();
16      }
17      else if(c>2){ //contradict to flag==1
18        target2();
19      }
20    }
21
22    if(d>3){
23      target3();
24    }
25  }
```

Huang et al. proposed to exploit the correlations between different targets [144], primarily in the form of path condition *overlapping*, *conflicting*, and *independence*, as illustrated in Figure 13. We use their running example in Listing 1 to show the three correlations [144]. (1) Condition a > 5 is an overlapping condition for both targets 1 and 2 because reaching them both requires the condition to hold. The overlapping condition can direct the fuzzer to cover the true branch at Line 5. (2) Reaching targets 1 and 2 have mutually exclusive demands on the condition of b, namely b $\leqslant$ 1 and b >1, which we refer to as a conflicting condition for the two targets. We regard the seeds satisfying b $\leqslant$ 1 as more likely to cover target 1 and seeds satisfying b > 1 are more likely to cover target 2. The fuzzer can more accurately select seeds for multiple targets by using conflict correlations to differentiate the difficulties in reaching different targets, for example, targets 1 and 2. (3) The condition d > 3 at Line 22 only influences the reachability of target 3 but does not affect whether targets 1 and 2 are reached. Similarly, condition a >5 affects reaching targets 1 and 2 but not target 3. As a result, we can mutate the independent bytes simultaneously, which could help approach multiple targets with fewer executions.

Herein, we suggest that the following relationships can be considered for DGF research.

The spatial relationship. Namely, the relative position of targets on the execution tree. Consider the relation between two targets, including whether they occupy the same branch, the level of shared executions, and their relative precedence if any.

The state relationship. For targets that involve the program state, consider their position in the state space. For example, whether two targets share the same state and whether two states can convert to each other on the state transition map.

The interleaving relationship. For multithreaded programs, thread scheduling also affects the execution ordering of events in different threads. Targets that can be reached under the same thread interleaving should have a close relationship in the interleaving space.

9.2 | Design multi-dimensional fitness metric

Current fuzzing approaches mainly focus on the coverage at the path level, such as maximizing the overall path coverage or reaching specific code parts, which neglects the fact that some bugs will not be triggered or manifest even when vulnerable code is exercised. For example, a buffer overflow vulnerability will be exhibited at a buffer access location only when the buffer access pointer points outside the buffer. Similarly, an integer overflow vulnerability will be observed at a program location only when the variable being incremented has a large enough value. To detect such "hard-to-manifest" vulnerability, the fitness metric must be extended to be multi-dimensional, such as the state space.

In practice, exploring a complex state machine is difficult, and most fuzzing-based approaches only make progress when exercising certain code, neglecting the update of the state machine and not fuzzing the corresponding test input further. However, some vulnerabilities may not be revealed for every visit to the program point. Only certain executions that reach the vulnerability point with the right state may exhibit vulnerable behavior. To expose such vulnerability, we need inputs that not only reach the vulnerability location but also match the vulnerable state [81].

$$headroom = \begin{cases} 0, & if\, A_c \geqslant A_h + s; \\ (A_h + s - A_c)/s, & if\, A_h \leqslant A_c < A_h + s; \\ 1, & otherwise. \end{cases} \tag{1}$$

In order to find hard-to-manifest vulnerability (e.g. buffer overflow and integer overflow), AFL-HR [81] defines a fitness metric ranging from 0 to 1, called *headroom*, to indicate how closely a test input can expose a potential vulnerability at a given vulnerability location. For example, for buffer overflow vulnerabilities, we consider the buffer access location is v_l, where *ptr* is the pointer, A_c is the value of *ptr* during the visit, A_h is the starting address of the allocated buffer and s is the size of the allocated buffer. As Equation (1) shows, the headroom is defined as the minimum distance between the location pointed to by the buffer access pointer and the end of the buffer across all visits to this location, divided by the size of the buffer.

IJON [29] leverages an annotation mechanism that allows a human analyst to help overcome roadblocks and explore the program's state space more thoroughly. Thus, state space is a dimension that is worth taking into account as a fitness metric alongside the reachability of the vulnerability location.

9.3 | Multi-objective optimization

For simplicity, the vast majority of contemporary studies have opted to ignore the possibility of multi-objective targeting through the simultaneous application of a range of metrics. For example, a tester might be interested in achieving higher coverage, while also targeting unusually long execution times, security properties, memory consumption, or energy consumption. Multi-objective optimization provides an advantage over traditional policies that are only capable of achieving one goal. It formulates the trade-off among multiple properties, such as usability and security [145]. For example, multi-objective optimization can generate test sets that cover specific targets while also maximizing overall coverage or prioritizing tests that cover as much of the software as possible while minimizing the amount of time that tests take to run [59]. The result of a multi-objective search is a set of Pareto-optimal solutions, where each member of the set is no better than any of the others for all of the objectives [59].

Multi-objective optimization is an open problem in the SBST community [59], which also is a challenge for DGF. A general solution for optimizing multiple objectives is co-evolution, where two (or more) populations of test inputs evolve simultaneously in a cooperative manner using their own fitness functions For example, AFL-HR [81] defines the fitness metric *headroom* to measure how closely a test input can expose a potential vulnerability at a given vulnerability location. Then it uses a co-evolutionary computation model to evolve test inputs for both coverage-based and headroom-based fitness metrics simultaneously. Similarly, other fitness metrics such as memory consumption [34] and new maxima of execution counts [37] have also been applied in a co-evolutionary manner. In contrast, FuzzFactory [99] provides a framework that supports multiple domain-specific objectives that are achieved by selecting and saving intermediate inputs from a custom predicate, which avoids the nontrivial implementation of mutation and search heuristics.

To show how multi-objective optimization works. We use MobFuzz [48] as an example. MobFuzz models the multi-objective optimization as the problem of selecting the best objective combination. The fuzzing process is divided into t intervals that each lasts for one minute. In the initial phase, the fuzzer selects the objective combinations in order and scores them by the rewards as Equation (2) defines [48].

$$\begin{aligned} Score(C_l,t) &= \overline{R}(C_l,t)+U(C_l,t) \\ &= \frac{\sum_{k=0}^{t} R(C_l,k)}{t}+\gamma * \sqrt{\frac{ln\left(\sum_{C_l \in C} n_l\right)}{n_l}} \end{aligned} \tag{2}$$

In Equation (2), MobFuzz applies the UCB1 [146] algorithm to select the combinations with the highest scores. In UCB1, $Score(C_l,t)$ denotes the final score for the combination to make decisions, which consists of two components: $\overline{R}(C_l,t)$ and $U(C_l,t)$. C denotes all the objective combinations, and C_l denotes the lth combination. $\overline{R}(C_l,t)$ is the average reward of C_l in previous t rounds, which gives combinations with greater historical rewards higher scores (exploitation). $R(C_l,k)$ is the reward of C_l in the kth round. $U(C_l,t)$ is the upper confidence bound of C_l, and it adds greater scores to combinations with smaller n_l values (the number of times the combination is selected), which is exploration. At the beginning of fuzzing, MobFuzz goes through an initial stage, in which each objective combination is selected once. After this stage, the n_l value of each combination will be 1. Next, at the end of each round of fuzzing, MobFuzz calculates the score of each combination and chooses the one with the maximum score as the objective combination for the next round. Moreover, γ is an empirical parameter in UCB1 that controls the balance between exploration and exploitation.

Then, it adaptively selects the objective combination that contains the most appropriate objectives for the current situation, the best objective combination with the highest reward is selected and given more power. Finally, an evolutionary algorithm is designed for multi-objective optimization in MobFuzz. The basic process is as follows: First, an initial population of seeds with a scale of N is selected. Next, the offspring seeds are obtained with crossover and mutation among the initial population. Then, execute the target program with each seed in the population and obtain related information. From the second generation onwards, the parent population and the offspring are combined to perform nondominated sorting. The seeds with updated objective values are selected to form a new parent population with a scale of N. Finally, the new offspring seeds are generated by the crossover and mutation among the new parent population. This process repeats until the predefined number of iterations is met.

To show the effectiveness of multi-objective optimization, we use metric *good seeds* to test MobFuzz and baseline fuzzers on four programs. Seeds that achieve greater objective values than the average in the selected objective combination are defined as *good seeds*. In Figure 14, the x-axis is the test time, and the y-axis is the percentages of good seeds generated. MobFuzz-N means MobFuzz disables multi-objective optimization. From the figure, we can conclude that by multi-objective optimization, the percentages of good seeds in MobFuzz are significantly greater than those of the baseline fuzzers.

9.4 | Target for new domains

Among the tools evaluated, only one (SemFuzz [22]) supports kernel code testing. Thus, introducing DGF to kernel code and guiding fuzzing towards critical sites such as syscalls and error handling codes to find kernel bugs should be a productive direction. Except for kernel testing, protocol testing is also suitable for DGF. Directed testing can strengthen the critical fields of the protocol message, such as the message length and control information. Zhu et al. [95] utilize DGF to construct more complete control flow graphs by targeting and exercising indirect jumps. It is delightful to see that DGF has been applied in the targeted testing for Register Transfer Level (RTL) designs [94]. Hopefully, DGF will be applied to more domains in the future.

Although DGF has been trying to discover new bug types, such as use-after-free and memory consumption bugs, many commonly seen bug types have not yet been included. Thus, another research direction is applying DGF to bug types with specific features, such as information leakage, time-of-check to time-of-use [147], and double-fetch bugs [116, 148]. For example, to detect a double-fetch bug, DGF would be useful to guide the testing towards code parts that launch continuous kernel reads of the same user memory address.

ACKNOWLEDGEMENTS

The authors would like to sincerely thank all the reviewers for their time and expertise on this paper. Your insightful comments help us improve this work. This work is partially supported by the National University of Defense Technology Research Project (ZK20-17 and ZK20-09), the National Natural Science Foundation China (62272472, U22B2005, and 61972412) and the HUNAN Province Natural Science Foundation (2021JJ40692).

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interest.

DATA AVAILABILITY STATEMENT

The authors confirm that the data supporting the findings of this study are available within the article.

ORCID

Pengfei Wang https://orcid.org/0000-0003-3408-4153

REFERENCES

1. Miller BP, Fredriksen L, So B. An empirical study of the reliability of unix utilities. Commun ACM. 1990;33(12):32–44.
2. Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M. The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng. 2019;47(11):2312–31.
3. Godefroid P. Fuzzing: hack, art, and science. Commun ACM. 2020;63(2):70–6.
4. Boehme M, Cadar C, Roychoudhury A. Fuzzing: challenges and reflections. IEEE Softw. 2021;38(3):79–86.
5. Zhu X, Wen S, Camtepe S, Xiang Y. Fuzzing: a survey for roadmap. ACM Comput Surv (CSUR). 2022;54(11s):1–36.
6. Liang H, Pei X, Jia X, Shen W, Zhang J. Fuzzing: state of the art. IEEE Trans Reliab. 2018;67(3):1199–218.
7. Li J, Zhao B, Zhang C. Fuzzing: a survey. Cybersecurity. 2018;1(1):1–13.
8. Blair W, Mambretti A, Arshad S, Weissbacher M, Robertson W, Kirda E, Egele M. Hotfuzz: discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing, 2020. arXiv preprint arXiv:2002.03416.
9. Joffe L, Clark D. Directing a search towards execution properties with a learned fitness function. In *the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019. p. 206–16.
10. Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T. KAFL: hardware-assisted feedback fuzzing for os kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017. p. 167–82.
11. Song D, Hetzelt F, Das D, Spensky C, Na Y, Volckaert S, et al. Periscope: an effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.
12. Kim K, Jeong DR, Kim CH, Jang Y, Shin I, Lee B. HFL: hybrid fuzzing on the linux kernel.
13. Yu B, Wang P, Yue T, Tang Y. Poster: fuzzing IoT firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. p. 2525–7.
14. Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. p. 1099–114.
15. Garbelini ME, Wang C, Chattopadhyay S. Greyhound: directed greybox wi-fi fuzzing. IEEE Trans Depend Sec Comput. 2020;19(2):817–34.
16. Shin Y, Williams L. Can traditional fault prediction models be used for vulnerability prediction? Empir Softw Eng. 2013;18(1):25–59.
17. Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, 2009. p. 474–84.
18. Ma K-K, Phang KY, Foster JS, Hicks M. Directed symbolic execution. In *International Static Analysis Symposium*, 2011. p. 95–111.
19. Person S, Yang G, Rungta N, Khurshid S. Directed incremental symbolic execution. Acm Sigplan Notices. 2011;46(6):504–15.
20. Marinescu PD, Cadar C. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013. p. 235–45.
21. Bohme M, Pham V-T, Nguyen M-D, Roychoudhury A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. p. 2329–44.
22. You W, Zong P, Chen K, Wang X, Liao X, Bian P, Liang B. Semfuzz: semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. p. 2139–54.
23. Peng J, Li F, Liu B, Xu L, Liu B, Chen K, Huo W. 1dVul: discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019. p. 605–16.
24. Nguyen M-D, Bardin S, Bonichon R, Groz R, Lemerre M. Binary-level directed fuzzing for use-after-free vulnerabilities, 2020. arXiv preprint arXiv:2002.10751.
25. Zhang J-M, Cui Z-Q, Chen X, Wu H-H, Zheng L-W, Liu J-B. Deltafuzz: historical version information guided fuzz testing. J Comput Sci Technol. 2022;37:29–49.
26. Zhu X, Bohme M. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021. p. 2169–82.
27. Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, Su P. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.
28. Kim J, Yun J. Poster: directed hybrid fuzzing on binary code. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. p. 2637–9.
29. Aschermann C, Schumilo S, Abbasi A, Holz T. Ijon: exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, 2020.
30. Li Y, Ji S, Lv C, Chen Y, Chen J, Gu Q, Wu C. V-fuzz: vulnerability-oriented evolutionary fuzzing, 2019. arXiv preprint arXiv:1901.01142.
31. Zhao Y, Li Y, Yang T, Xie H. Suzzer: a vulnerability-guided fuzzer based on deep learning. In *International Conference on Information Security and Cryptology*, 2019. p. 134–53.
32. Zhu X, Liu S, Li X, Wen S, Zhang J, Seyit C, Xiang Y. Defuzz: deep learning guided directed fuzzing, 2020. arXiv preprint arXiv:2010.12149.
33. Osterlund S, Razavi K, Bos H, Giuffrida C. Parmesan: sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
34. Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, et al. Memlock: memory usage guided fuzzing. In *ICSE*, 2020.
35. Wang H, Xie X, Li Y, Wen C, Liu Y, Qin S, et al. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*: Seoul, South Korea, 2020.
36. Kim T, Kim CH, Rhee J, Fei F, Tu Z, Walkup G, et al. Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. p. 425–42.

37. Lemieux C, Padhye R, Sen K, Song D. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018. p. 254–65.
38. Petsios T, Zhao J, Keromytis AD, Jana S. Slowfuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. p. 2155–68.
39. Feng X, Sun R, Zhu X, Xue M, Wen S, Liu D, et al. Snipuzz: black-box fuzzing of IoT firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021. p. 337–50.
40. Gascon H, Wressnegger C, Yamaguchi F, Arp D, Rieck K. Pulsar: stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, 2015. p. 330–47.
41. Shu Z, Yan G. IoTInfer: automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference. IEEE Internet Things J. 2022;9:22737–51.
42. De Ruiter J, Poll E. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015. p. 193–206.
43. Zalewski M. American fuzzy lop, 2020. http://lcamtuf.coredump.cx/afl/
44. Bohme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. IEEE Trans Softw Eng. 2017;45(5):489–506.
45. Lyu C, Ji S, Zhang C, Li Y, Lee W-H, Song Y, Beyah R. MOPT: optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. p. 1949–66.
46. Chen P, Liu J, Chen H. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019. p. 499–513.
47. Yue T, Tang Y, Yu B, Wang P, Wang E. Learnafl: Greybox fuzzing with knowledge enhancement. IEEE Access. 2019;7:117029–43.
48. Zhang G, Wang P, Yue T, Kong X, Huang S, Zhou X, Lu K. Mobfuzz: adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.
49. Lemieux C, Sen K. Fairfuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage, 2017. arXiv preprint arXiv:1709.07101.
50. Godefroid P, Levin MY, Molnar DA. Automated whitebox fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium*, vol. 8, 2008. p. 151–66.
51. Cadar C, Dunbar D, Engler DR. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs, vol. 8, OSDI, 2008. p. 209–24.
52. Jin W, Orso A. Bugredux: reproducing field failures for in-house debugging. In *the 34th International Conference on Software Engineering (ICSE)*, 2012. p. 474–84.
53. Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013. p. 49–64.
54. Bohme M, Oliveira BrunoCDS, Roychoudhury A. Partition-based regression verification. In *the 35th International Conference on Software Engineering (ICSE)*, 2013. p. 302–11.
55. Santelices R, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008. p. 218–27.
56. Xu Z, Kim Y, Kim M, Rothermel G, Cohen MB. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010. p. 257–66.
57. Rossler J, Zeller A, Fraser G, Zamfir C, Candea G. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013. p. 114–23.
58. McMinn P. Search-based software test data generation: a survey. Softw Test Verification Reliab. 2004;14(2):105–56.
59. McMinn P. Search-based software testing: past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011. p. 153–63.
60. Miller W, Spooner DL. Automatic generation of floating-point test data. IEEE Trans Softw Eng. 1976;3:223–6.
61. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. Inform Softw Technol. 2001;43(14):841–54.
62. Buehler O, Wegener J. Evolutionary functional testing of an automated parking system. In *Proceedings of the International Conference on Computer, Communication and Control Technologies (CCCT'03) and the 9th. International Conference on Information Systems Analysis and Synthesis (ISAS'03), Florida, USA*, 2003.
63. Buhler O, Wegener J. Evolutionary functional testing. Comput Oper Res. 2008;35(10):3144–60.
64. Puschner P, Nossal R. Testing the results of static worst-case execution-time analysis. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, 1998. p. 134–43.
65. Wegener J, Sthamer H, Jones BF, Eyres DE. Testing real-time systems using genetic algorithms. Softw Qual J. 1997;6(2):127–35.
66. Wegener J, Grochtmann M. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst. 1998;15(3): 275–98.
67. Schultz AC, Grefenstette JJ, De Jong KA. Test and evaluation by genetic algorithms. IEEE Expert. 1993;8(5):9–14.
68. Briand LC, Feng J, Labiche Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, 2002. p. 43–50.
69. Colanzi TE, Assunccao WKG, Vergilio SR, Pozo A. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In *International Symposium on Search Based Software Engineering*, 2011. p. 188–203.
70. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. IEEE Trans Softw Eng. 2007;33(4):225–37.
71. Briand LC, Labiche Y, Shousha M. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, 2005. p. 1021–8.
72. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008. p. 249–58.
73. Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering, 2003. Proceedings*, 2003. p. 38–48.
74. Petke J, Yoo S, Cohen MB, Harman M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013. p. 26–36.
75. Cohen MB, Dwyer MB, Shi J. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007. p. 129–39.

76. Derderian K, Hierons RM, Harman M, Guo Q. Automated unique input output sequence generation for conformance testing of fsms. Comput J. 2006;49(3):331–44.
77. Derderian KA. Automated test sequence generation for finite state machines using genetic algorithms. *Ph.D. Thesis*, Oxford,UK, 2006.
78. Lehre PK, Yao X. Runtime analysis of the (1+ 1) ea on computing unique input output sequences. Inf Sci. 2014;259:510–31.
79. Tracey N, Clark J, Mander K, McDermid J. Automated test-data generation for exception conditions. Softw Pract Experience. 2000;30(1): 61–79.
80. Tracey N, Clark J, McDermid J, Mander K. A search-based automated test-data generation framework for safety-critical systems. In Systems engineering for business process change: new directions Springer; 2002. p. 174–213.
81. Medicherla RK, Komondoor R, Roychoudhury A. Fitness guided vulnerability detection with greybox fuzzing. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020. p. 513–20.
82. Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. p. 2095–108.
83. Liang H, Zhang Y, Yu Y, Xie Z, Jiang L. Sequence coverage directed greybox fuzzing. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019. p. 249–59.
84. Situ L, Wang L, Li X, Guan L, Zhang W, Liu P. Energy distribution matters in greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019. p. 270–1.
85. Wustholz V, Christakis M. Targeted greybox fuzzing with static lookahead analysis, 2019. arXiv preprint arXiv:1905.07147.
86. Ji T, Wang Z, Tian Z, Fang B, Ruan Q, Wang H, Shi W. AFLPro: direction sensitive fuzzing. J Inform Secur Appl. 2020;54:102497.
87. Liang H, Jiang L, Ai L, Wei J. Sequence directed hybrid fuzzing. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020. p. 127–37.
88. Ye J, Li R, Zhang B. RDFuzz: accelerating directed fuzzing with intertwined schedule and optimized mutation. Math Probl Eng. 2020;2020: 1–12.
89. Wang Z, Liblit B, Reps T. TOFU: target-orienter fuzzer, 2020. arXiv preprint arXiv:2004.14375.
90. Li R, Liang H, Liu L, Ma X, Qu R, Yan J, Zhang J. GTFuzz: guard token directed grey-box fuzzing. In *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2020. p. 160–70.
91. Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K. Fuzzguard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning, 2020.
92. Huang H, Guo Y, Shi Q, Yao P, Wu R, Zhang C. Beacon: directed grey-box fuzzing with provable path pruning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.
93. Lee G, Shim W, Lee B. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
94. Canakci S, Delshadtehrani L, Eris F, Taylor MB, Egele M, Joshi A. DirectFuzz: automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.
95. Zhu K, Lu Y, Huang H, Yu L, Zhao J. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. Appl Scie. 2021; 11(3):1351.
96. Wang S, Jiang X, Yu X, Sun S. KCFuzz: directed fuzzing based on keypoint coverage. In *International Conference on Artificial Intelligence and Security*, 2021. p. 312–25.
97. Du Z, Li Y, Liu Y, Mao B. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.
98. Jain V, Rawat S, Giuffrida C, Bos H. TIFF: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018. p. 505–17.
99. Padhye R, Lemieux C, Sen K, Simon L, Vijayakumar H. Fuzzfactory: domain-specific fuzzing with waypoints. Proc ACM Program Lang. 2019;3(OOPSLA):1–29.
100. Chen Y, Li P, Xu J, Guo S, Zhou R, Zhang Y, et al. Savior: towards bug-driven hybrid testing, 2019. arXiv preprint arXiv:1906.07327.
101. Medicherla RK, Nagalakshmi M, Sharma T, Komondoor R. HDR-Fuzz: detecting buffer overruns using addresssanitizer instrumentation and fuzzing, 2021. arXiv preprint arXiv:2104.10466.
102. Li P, Liu Y, Meng W. Understanding and detecting performance bugs in markdown compilers. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.
103. Christakis M, Muller P, Wustholz V. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, 2016. p. 144–55.
104. Du X, Chen B, Li Y, Guo J, Zhou Y, Liu Y, Jiang Y. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019. p. 60–71.
105. 9 documentation c. Undefined behavior sanitizer, 2020. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
106. zynamics.com. Bindiff, 2020. https://www.zynamics.com/bindiff.html
107. Du X. Towards building a generic vulnerability detection platform by combining scalable attacking surface analysis and directed fuzzing. In *International Conference on Formal Engineering Methods*, 2018. p. 464–8.
108. Mathis B, Gopinath R, Mera M, Kampmann A, Hoschele M, Zeller A. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019. p. 548–60.
109. Chen Y, Jiang Y, Ma F, Liang J, Wang M, Zhou C, et al. Enfuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019. p. 1967–83.
110. Liang J, Jiang Y, Chen Y, Wang M, Zhou C, Sun J. PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018. p. 809–14.
111. Luo C, Meng W, Li P. Selectfuzz: efficient directed fuzzing with selective path exploration. In *IEEE Symposium on Security and Privacy (SP)*, IEEE; 2023. p. 2693–707.
112. Chen Y, Mu D, Xu J, Sun Z, Shen W, Xing X, et al. PTRIX: efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019. p. 633–45.
113. Zhang G, Zhou X, Luo Y, Wu X, Min E. Ptfuzz: guided fuzzing with processor trace feedback. IEEE Access. 2018;6:37302–13.
114. Swiecki R. Honggfuzz, 2016. Available online at: http://code.google.com/p/honggfuzz

115. Wang P, Krinke J, Zhou X, Lu K. Avpredictor: comprehensive prediction and detection of atomicity violations. Concurr Comput: Pract Experience. 2019;31(15):e5160.
116. Wang P, Krinke J, Lu K, Li G, Dodier-Lazaro S. How double-fetch situations turn into double-fetch vulnerabilities: a study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017. p. 1–16.
117. Mu D, Cuevas A, Yang L, Hu H, Xing X, Mao B, Wang G. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018. p. 919–36.
118. Fell J. A review of fuzzing tools and methods. 2017. PenTest Magazine.
119. Böhme M, Cadar C, Roychoudhury A. Fuzzing: challenges and reflections. IEEE Softw. 2020;38(3):79–86.
120. Eisele M, Maugeri M, Shriwas R, Huth C, Bella G. Embedded fuzzing: a review of challenges, tools, and solutions. Cybersecurity. 2022;5(1):18.
121. Saavedra GJ, Rodhouse KN, Dunlavy DM, Kegelmeyer PW. A review of machine learning applications in fuzzing, 2019. arXiv preprint arXiv:1906.11133.
122. Wang Y, Jia P, Liu L, Huang C, Liu Z. A systematic review of fuzzing based on machine learning techniques. PloS one. 2020;15(8):e0237749.
123. Zhang Y, Zhang J, Zhang D, Mu Y. Survey of directed fuzzy technology. In *2018 ieee 9th international conference on software engineering and service science (icsess)*, IEEE; 2018. p. 1–4.
124. Nguyen HL, Grunske L. BeDivFuzz: integrating behavioral diversity into generator-based fuzzing. In *Proceedings of the 44th international conference on software engineering*, ACM; 2022. p. 249–61.
125. Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. In *Proceedings of the 29th acm sigplan conference on programming language design and implementation*, ACM; 2008. p. 206–15.
126. Padhye R, Lemieux C, Sen K, Papadakis M, Le Traon Y. Semantic fuzzing with zest. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*, ACM; 2019. p. 329–40.
127. Tech P. Peach fuzzer platform, 2020. https://www.peach.tech/products/peach-fuzzer/peach-platform/
128. Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In *21st usenix security symposium (usenix security 12)*, USENIX; 2012. p. 445–58.
129. Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi A-R, Teuchert D. Nautilus: fishing for deep bugs with grammars. In *NDSS*, 2019.
130. Jero S, Pacheco ML, Goldwasser D, Nita-Rotaru C. Leveraging textual specifications for grammar-based fuzzing of network protocols. In *Proceedings of the aaai conference on artificial intelligence*, vol. 33, AAAI, 2019. p. 9478–83.
131. Xu W, Park S, Kim T. Freedom: engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 acm sigsac conference on computer and communications security*, ACM; 2020. p. 971–86.
132. Park S, Xu W, Yun I, Jang D, Kim T. Fuzzing javascript engines with aspect-preserving mutation. In *2020 ieee symposium on security and privacy (sp)*, IEEE; 2020. p. 1629–42.
133. Fratric I. Dom fuzzer, 2017. https://github.com/googleprojectzero/domato
134. Wang J, Chen B, Wei L, Liu Y. Skyfire: data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017. p. 579–94.
135. Liu X, Li X, Prajapati R, Wu D. DeepFUZZ: automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the aaai conference on artificial intelligence*, vol. 33, AAAI, 2019. p. 1044–51.
136. Godefroid P, Peleg H, Singh R. Learn&fuzz: machine learning for input fuzzing. In *2017 32nd ieeelacm international conference on automated software engineering (ase)*, IEEE; 2017. p. 50–9.
137. Whitehouse O. Introduction to anti-fuzzing: a defence in depth aid, 2014. https://www.nccgroup.trust/sg/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/
138. Edholm E, Göransson D. 2016. Escaping the fuzz-evaluating fuzzing techniques and fooling them with anti-fuzzing.
139. Jung J, Hu H, Solodukhin D, Pagan D, Lee KH, Kim T. Fuzzification:{Anti-Fuzzing} techniques. In *28th Usenix Security Symposium (Usenix Security 19)*, USENIX Association; 2019. p. 1913–30.
140. Güler E, Aschermann C, Abbasi A, Holz T. {AntiFuzz}: impeding fuzzing audits of binary executables. In *28th Usenix Security Symposium (Usenix Security 19)*, USENIX Association; 2019. p. 1931–47.
141. Li Y, Meng G, Xu J, Zhang C, Chen H, Xie X, Wang H, Liu Y. Vall-nut: principled anti-grey box-fuzzing. In *2021 ieee 32nd International Symposium on Software Reliability Engineering (issre)*, IEEE; 2021. p. 288–99.
142. Zhou Z, Wang C, Zhao Q. No-fuzz: efficient anti-fuzzing techniques. In *International conference on security and privacy in communication systems*, Springer; 2022. p. 731–51.
143. Jiang M, Xu T, Zhou Y, Hu Y, Zhong M, Wu L, Luo X, Ren K. Examiner: automatically locating inconsistent instructions between real devices and cpu emulators for arm. In *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*, ACM; 2022. p. 846–58.
144. Huang H, Yao P, Chiu H-C, Guo Y, Zhang C. Titan: efficient multi-target directed greybox fuzzing.
145. Harman M, Jia Y, Zhang Y. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015. p. 1–12.
146. Agrawal R. Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. Adv Appl Probab. 1995;27(4):1054–78.
147. Wei J, Pu C. Tocttou vulnerabilities in unix-style file systems: an anatomical study., vol. 5, FAST; 2005. p. 12–12.
148. Wang P, Lu K, Li G, Zhou X. A survey of the double-fetch vulnerabilities. Concurr Computat: Pract Experience. 2018;30(6):e4345.

How to cite this article: Wang P, Zhou X, Yue T, Lin P, Liu Y, Lu K. The progress, challenges, and perspectives of directed greybox fuzzing. Softw Test Verif Reliab. 2023;e1869. https://doi.org/10.1002/stvr.1869