TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers

Sadullah Canakci , Nikolay Matyunin, Kalman Graffi, Manuel Egele

2022

Preprint:

TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers

Sadullah Canakci
scanakci@bu.edu
Boston University
Boston, MA, USA

Nikolay Matyunin
nikolay.matyunin@honda-ri.de
Honda Research Institute Europe GmbH
Offenbach, Germany

Kalman Graffi
Kalman.Graffi@honda-ri.de
Honda Research Institute Europe GmbH
Offenbach, Germany

Ajay Joshi
joshi@bu.edu
Boston University
Boston, MA, USA

Manuel Egele
megele@bu.edu
Boston University
Boston, MA, USA

ABSTRACT

Software development is a continuous and incremental process. Developers continuously improve their software in small batches rather than in one large batch. The high frequency of small batches makes it essential to use effective testing methods that detect bugs under limited testing time. To this end, researchers propose directed greybox fuzzing (DGF) which aims to generate test cases towards stressing certain target sites. Different from the coverage-based greybox fuzzing (CGF) which aims to maximize code coverage in the whole program, the goal of DGF is to cover potentially buggy code regions (e.g., a recently modified program region). While prior works improve several aspects of DGF (such as power scheduling, input prioritization, and target selection), little attention has been given to improving the seed selection process. Existing DGF tools use seed corpora mainly tailored for CGF (i.e., a set of seeds that cover different regions of the program). We observe that using CGF-based corpora limits the bug-finding capability of a directed greybox fuzzer. To mitigate this shortcoming, we propose TargetFuzz, a mechanism that provides a DGF tool with a target-oriented seed corpus. We refer to this corpus as DART corpus, which contains only 'close' seeds to the targets. This way, DART corpus guides DGF to the targets, thereby exposing bugs even under limited fuzzing time. Evaluations on 34 real bugs show that AFLGo (a state-of-the-art directed greybox fuzzer), when equipped with DART corpus, finds 10 additional bugs and achieves 4.03× speedup, on average, in the time-to-exposure compared to a generic CGF-based corpus.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

directed greybox Fuzzing; patch testing; seed selection strategy; coverage-based greybox fuzzing; seed corpus

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3501276

ACM Reference Format:
Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. 2022. TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3488932.3501276

1 INTRODUCTION

The nature of modern software development is continuous and incremental. Rather than in one large batch, developers continuously build, test, and deploy their changes in small batches. Developers use effective testing techniques to detect bugs as software evolves. Due to its ease in deployment and effectiveness in bug finding, it has become a common practice for developers to use Greybox Fuzzing (GF) as part of their testing framework. After each pull request or commit, it is common to run some fuzzing sessions with the goal of finding bugs before integrating submitted changes into the codebase.

To date, researchers have proposed two types of GF techniques: Coverage-based Greybox Fuzzing (CGF) and Directed Greybox Fuzzing (DGF). The goal of CGF is to maximize the code coverage of the program, thereby discovering bugs anywhere in the program. When the program undergoes substantial changes, CGF is effective, especially with long fuzzing runs. However, it is rare to observe substantial software changes in a real-world software project. The maintenance of real-world projects often involves frequent commits where each commit only modifies a few lines of code. As a reference point, the average commit size across five popular projects (i.e., PHP, Libxml2, Openssl, SQLite3, and Libpng) is 34.8 lines of code [1]. Moreover, fuzzing requires resources and hence is subject to budget constraints, especially time. The commit frequency of popular projects is generally high (on average one commit for every 20.4 hours for the aforementioned five projects) and in turn results in limited fuzzing time. A well-suited fuzzing technique for testing small size commits, especially under limited fuzzing time, is DGF which steers the test generation towards specific target sites (e.g., code lines modified by a recent commit) rather than unrelated program components. DGF computes the distance of each seed with respect to target sites by averaging the weight of executed basic blocks. The weight of each basic block is determined based on the shortest paths to target basic blocks in the program

[1] We use the master branch of each project and include commits that only change source code (including both additions and deletions) in the last 14 years.

inter-procedural control-flow graph. During fuzzing, DGF focuses on closer seeds (i.e., seeds with lower distances) to cover modified code regions, thereby increasing the likelihood of detecting bugs even under limited fuzzing time.

While prior works improve several aspects of DGF (such as energy assignment [7], seed scheduling [51], and target selection [36, 42]), little attention has been given to improving the seed selection process that determines seeds in an initial seed corpus. The initial seed corpus consists of a set of valid inputs (e.g., png files for a png processor tool) serving as starting points to the fuzzer. The seed selection process aims to identify high-quality seeds from a large seed pool that maximize the effectiveness of a fuzzer (mainly measured with its bug finding capability). If an initial seed corpus is not provided, the fuzzer wastes the limited fuzzing time to infer the file format that the program under test (PUT) accepts. In case a seed corpus is provided, the quality and the quantity of seeds significantly impacts the effectiveness of the fuzzer in bug detection [27, 39]. Although there exists a variety of seed selection strategies/tools [25, 31, 39, 49], they are mainly designed for CGF. In this work, we tackle the question of whether a seed corpus tailored for DGF outperforms a CGF-based seed corpus when used with a DGF tool.

To date, all the academic DGF works [3, 4, 7, 9, 10, 30, 35, 36, 42, 44, 51] equip their directed greybox fuzzers with a seed corpus designed for CGF (CGF-based seed corpus) for the evaluation. Specifically, these fuzzers commonly use regression tests [20, 40], generic seeds [14, 17], and generic dictionaries [16] as part of a seed corpus. Additionally, they extensively use a variety of seed selection strategies [25, 31, 39, 49] to ensure that only high quality seeds construct their seed corpora. At their core, these strategies select a minimal set of inputs (as part of a seed corpus) that provide maximal code coverage and therefore are designed for CGF.

We make a key observation that using CGF-based seed corpora significantly limits the bug detection capability of DGF. Moreover, we observe that a seed selection mechanism (namely `MINSET` [39]) that is well-suited for CGF performs poorly with DGF. These observations are intuitive. A seed corpus tailored for CGF exercises different regions of the whole program, thereby triggering bugs in different parts of the program. However, DGF aims to reach a specific region (e.g., newly written or modified code) of the program and trigger bugs in that region. When DGF uses a seed corpus that covers different regions of the program, DGF wastes part of its effort by spending time on fuzzing "farther away" seeds that are unrelated to the target regions. Instead, DGF should be provided with a seed corpus that contains "closer" seeds to the target sites so that DGF spends the limited testing time on fuzzing seeds related to targets.

In this work, we present TargetFuzz, a mechanism that provides a target-specific seed corpus to a DGF tool. At its core, TargetFuzz exploits the continuous and incremental nature of software development. For the frequent mode of software development (i.e., the mode where small-size changes happen), TargetFuzz outputs a commit-specific seed corpus which we refer to as DART corpus[2]. TargetFuzz uses a seed selection strategy that accounts for the seed distances rather than only code coverage to select seeds in DART corpus. The strategy selects a subset of seeds that are 'close' to the modified target regions from a large seed pool. A DGF tool equipped with DART corpus spends most of the fuzzing time budget on stressing modified code regions rather than unrelated program components. In this way, the DGF tool achieves better code coverage and successfully exposes bugs in target sites even under limited testing time. TargetFuzz uses the rare mode of software development (i.e., the mode where substantial software changes happen) to generate the large seed pool. Using coverage-based fuzzers, TargetFuzz generates a variety of seeds that exercise different regions of the program. Whenever a developer submits a small-size commit, TargetFuzz selects those seeds that are related to modified code region from the large seed pool as opposed to a CGF-based corpus that covers unrelated code regions as well.

We test TargetFuzz with the state-of-the-art directed greybox fuzzer (AFLGo [4]) using a fuzzing benchmark (Magma [24]). In total, we test 34 vulnerabilities on 7 popular programs. Experimental results show that AFLGo, when equipped with DART corpus, triggers 10 more unique bugs (out of 34) and achieves 4.03× speedup, on average, in the time-to-exposure (TTE) of bugs compared to a generic corpus. We also compare TargetFuzz's distance-based seed selection strategy with a seed selection strategy tailored for CGF (`MINSET` [39]). When equipped with DART, AFLGo discovers 8 more unique bugs (out of 34) compared to `MINSET`-based corpus and triggers the bugs 30% faster (on average).

Overall, the main contributions of this work are as follows:

- We are first to observe that CGF-based seed corpora consisting of generic seeds limit the effectiveness of a DGF tool. Based on this observation, we design TargetFuzz, a mechanism that provides a DGF tool with different target-specific seed corpora for different fuzzing targets.
- We show that the state-of-the-art CGF-based seed selection strategies are not well-suited for DGF. To increase the bug detection capability of a DGF tool, TargetFuzz proposes a seed selection strategy that takes seed distance metric into account rather than merely code coverage.
- We evaluate TargetFuzz on a variety of real bugs in real-world programs (in total 34 bugs across 7 libraries). Experimental results show that a DGF tool can find more unique bugs when equipped with a seed corpus provided by TargetFuzz compared to a corpus tailored for CGF.
- In the spirit of open science and to facilitate reproducibility of our experiments, we will make our data set and source code of TargetFuzz publicly available.

2 BACKGROUND

In this section, we first provide a background on CGF. Next, we explain the major differences between CGF and DGF.

2.1 Coverage-based Greybox Fuzzing

The main goal of CGF is to maximize the code coverage of a PUT. The intuition is that executing different code blocks in the target program with different inputs increases the chance of exposing bugs. At a high level, CGF generates a large number of test inputs by performing a set of mutation operations (such as bit flips) and

[2]We use DART to signify that our corpus is targeted to a (code) region and needs to achieve its goal (covering that particular code region) quickly similar to a dart that can be thrown at a target with a quick movement.

Algorithm 1: Coverage-based Greybox Fuzzing

```
(S1) Input  : Initial Seed Corpus S, time_limit
Output : Crashing Input Set CI
CI ← ∅;
while time_elapsed < time_limit do
    (S2) s ← ScheduleNext(S);
    (S3) e ← AssignEnergy(s);
    for i = 1 to e do
        (S4) m' = MutateInput(s);
        (S5) o = ExecutePUT(m');
        (S6) if is_CRASHING(o) then
                 add m' to CI;
        (S6) else if is_INTERESTING(o) then
                 add m' to S;
    end
end
return CI
```

executes the PUT with the generated inputs. Based on the coverage-feedback recorded from the program at runtime, CGF determines the 'interesting' inputs and uses them for generating a new set of inputs. Here an 'interesting' input refers to any input that increases code coverage.

We provide an algorithmic sketch of CGF in Algorithm 1. In stage **S1**, a fuzzer takes a seed corpus and a time limit as inputs. A seed corpus includes an initial set of test inputs which serve as starting points to the fuzzer. Seeds usually have infinite domains. For instance, a png processor tool can be provided with infinite number of valid png files. To reduce the size of the initial seed corpus, it is a common practice to use a seed selection strategy which selects a subset of seeds from a large seed pool. TargetFuzz mainly focuses on this aspect of fuzzing with more details provided in Section 4. Besides a seed corpus, stage **S1** takes as input a time limit that determines the total duration for fuzzing. Once fuzzing starts, all the remaining stages (i.e., **S2-S6**) repeat in a loop until the time limit is reached. Specifically, stage **S2** picks a seed from the seed corpus for the next fuzzing iteration. This stage of fuzzing is referred to as seed scheduling. The goal of seed scheduling is to pick an input that is more likely to increase coverage with mutations. Stage **S3** assigns energy to the scheduled seed. Energy of the seed determines the total number of mutations. Fuzzer performs more mutations on seeds with higher energy levels. Stage **S4** mutates the scheduled seed to produce a new set of seeds. This stage is referred to as seed generation. Stage **S5** executes the PUT with a mutated seed and makes an observation. Stage **S6** checks the observation for the mutated seed to determine if the seed increases coverage of the program (i.e., interesting) or results in a crash. Stage **S6** adds interesting seeds to the seed corpus and crashing seeds to a separate crashing seed set.

2.2 Directed Greybox Fuzzing

In a software project, any commit that changes a part of program source produces a candidate buggy code region. The fuzzer should aim to maximize code coverage of the potential buggy code region rather than the whole program. DGF [4] is a suitable approach when this target program region is known. When a new commit is submitted by a developer, DGF can be used to generate test cases towards stressing target sites modified by the commit. During fuzzing, DGF steers the input generation towards modified code regions to detect any bugs in the target region.

Table 1: Seed corpora used by DGF tools. RT refers to regression tests, GIT refers to seeds obtained from a git repo, N/A states that the origin of seed corpus is not clearly specified.

Tool Name	Seed Corpus
AFLGo [4]	RT, OSS-Fuzz Seeds, Generic Seeds (AFL)
Hawkeye [7]	Same as AFLGo
LOLLY [30]	RT, Generic Seeds (AFL)
Memfuzz [10]	Null Seed, RT, Generic Dictionary (AFL)
TortoiseFuzz [44]	GIT, RT, Generic Seeds (N/A)
UAFuzz [35]	Null Seed, Generic Seeds (N/A)
UAFL [42]	RT, Random Seeds from the internet
Memlock [45]	N/A
IJON [3]	Seed Containing 'a', Generic Seeds (random) Dictionary of Strings from Source
FuzzGuard [51]	Same as AFLGo
ParmeSan [36]	Single file with '\n', Google Fuzzer Test-suite
1DVUL [38]	N/A
SAVIOR [9]	Generic Seeds (AFL), RT
CAFL [28]	N/A

DGF and CGF mainly differ at stage **S3** (i.e., how long to fuzz the chosen seed). The energy value of a seed controls the total number of mutations that need to be applied to the seed (i.e., higher energy results in more seed mutations). CGF [5, 18, 21, 31, 48] determines the energy of a seed with the goal of exploring more code regions. For instance, AFL [18] assigns more energy to those seeds that exercise a new path. DGF [4, 7] takes the program structure into account in its energy assignment formula so that it can spend longer fuzzing time on seeds which are likely to cover certain target sites. To achieve this goal, DGF instruments the source code by analyzing call graph (CG) and control-flow graphs (CFGs) of the program. First, DGF assigns a value (function-level target distance) to each node in the CG on function-level by finding the shortest-path of each node to the target function (e.g., modified function). Next, DGF computes the basic-block-level target distance of each basic block to all basic blocks that call a function in the CFG. Function-level target distance and basic-block-level target distance together define a seed distance formula. At run-time, fuzzer uses the exercised basic blocks in the execution trace along with the seed distance formula to compute the seed distance. DGF determines energy of a seed based on the seed distance and performs more mutations on a seed that is 'closer' (i.e., has lower seed distance) than on a seed that is 'further away' from target sites. The intuition is that the seeds mutated from a close seed are more likely to reach the target sites. Note that some of the DGF works [7, 36] utilize seed distances at stage **S2** to prioritize closer seeds in seed scheduling.

In Table 1, we provide several DGF works [3, 4, 7, 9, 10, 28, 30, 35, 36, 42, 44, 45, 51] with their corresponding initial seed corpus. As it is evident from the table, prior DGF works use seed corpora tailored for CGF such as generic seeds provided by AFL, randomly crawled seeds from the Internet, regression tests, etc. In our experiments, we observe that a corpus that is well-suited for CGF can limit the bug-finding capability of DGF. To this end, TargetFuzz uses a seed

selection strategy that is suited for DGF and provides a target-specific seed corpus to a DGF tool in stage **S1**. Our mechanism takes seed distance into account rather than only coverage and helps a DGF tool to spend more time on fuzzing seeds related to the target sites.

3 RELATED WORK

In this section, we present prior works that are related to TargetFuzz from three different research angles; the seed selection strategies, generation-based fuzzers, and methods that enhance the bug finding capability of greybox fuzzers.

3.1 Seed Selection Strategies

TargetFuzz provides a target-specific seed corpus to a DGF tool by selecting a subset of seeds from a large seed pool. Therefore, it is highly related to prior seed selection strategies that reduce the size of a large seed pool. In Table 2, we provide a brief summary of different seed selection strategies. While the main goal of prior seed selection strategies is to select a subset of seeds that cover all the unique edges in a seed pool, TargetFuzz aims to select only "closer" seeds to the target region. This way, TargetFuzz aims to guide a DGF tool to the target regions.

`afl-cmin` [49] is one of the widely-used tools that relies on AFL's own notion of edge coverage when selecting a subset of seeds. `afl-cmin` first computes the edge coverage (a.k.a. branch coverage) of each seed in the large seed pool. Next, it finds a set of edges by calculating the union of edge coverage across all the seeds. To reduce the corpus size, `afl-cmin` performs a greedy search that selects the smallest size of seed for each edge in the set.

Abdelnur et al. [1] propose a seed selection strategy (`MINSET`) that applies the minimum set cover (MSC) problem to the edge covering. Since MSC is an NP-hard problem, the authors use an approximation greedy algorithm. In Algorithm 2, we provide the greedy algorithm for `MINSET`. Given a union set of covered edges U and a list of sets S where each set contains covered edges from a particular seed, `MINSET` computes a minimum set C that covers all the edges in U. To compute C, at each iteration, `MINSET` greedily searches for the set S_i that contains the greatest number of uncovered elements by C. Later, Rebert et al. [39] extend MSC to weighted set cover problem by considering seed execution times (`TIMEMINSET`) and seed sizes (`SIZEMINSET`). Among these three approaches, `MINSET` has the highest bug-finding detection capability as shown by Rebert et al. [39].

A recent work by Herrera et. al [25] proposes a new seed selection strategy, namely `OPTIMIN`, that implements an optimal corpus minimization tool for AFL. Seed selection strategies like `TIMEMINSET` and `SIZEMINSET` employ some heuristics to compute a set cover `C` since the underlying problem (i.e., weighted set cover) is NP-complete. `OPTIMIN` computes the exact solution to this NP-complete problem in a reasonable time by interpreting the problem as a maximum satisfiability (MaxSAT) problem. The MaxSAT relies on hard and soft constraints where the goal is to satisfy all hard constraints and maximize the number of satisfied soft constraints. By treating edge coverage as a hard constraint while excluding a seed in the solution as a soft constraint, `OPTIMIN` guarantees to cover all edges with the minimal number of seeds.

Table 2: Comparison of different seed selection strategies.

Strategy	Goal	Approach	Target
afl-cmin [49]	Cover all edges in the seed pool	Perform greedy search to select the smallest size of seeds	CGF
MINSET [1]	Cover all edges in the seed pool	Perform greedy search to select seeds covering greatest number of uncovered edges	CGF
TIMEMINSET [39]	Cover all edges in the seed pool	Extend MINSET's approach using seed execution times	CGF
SIZEMINSET[39]	Cover all edges in the seed pool	Extend MINSET's approach using seed sizes	CGF
OPTIMIN [25]	Cover all edges in the seed pool	Solve seed selection as a MaxSat problem	CGF
TargetFuzz (this work)	Select 'close' seeds from the seed pool	Select the seeds with lowest seed distances	DGF

Algorithm 2: Greedy Algorithm for MINSET

Input : A Set of Covered Branches U
Input : A List of Sets S
Output: Set Cover C
$C \leftarrow \emptyset$;
while *U contains elements not covered by C* **do**
  Find the set S_i containing the greatest number of uncovered elements
  Add S_i to C
end
return C

At its core, all the five aforementioned seed selection strategies aim to maximize the code coverage. We demonstrate (in Section 6) that these coverage-based seed selection strategies are not well-suited for DGF (i.e., DGF wastes the fuzzing effort on unrelated code) and propose a seed selection strategy that accounts for seed distance rather than merely coverage.

3.2 Generation-based Fuzzers

Another line of research attempts to generate seeds using models such as a grammar that characterizes an input format. Several generation-based fuzzers produce test cases by using a predefined model or a model inferred from the program. The model can target a specific language or a grammar that defines input format constraints for the PUT. Specifically, IFuzzer [41] and LangFuzz [26] generate JavaScript inputs by randomly recombining extracted code fragments from a set of given seeds. BlendFuzz [47] uses the similar approach for XML and regular expression parsers. Skyfire [43] learns a probabilistic context-sensitive model from the program with a data-driven static analysis and uses the inferred model to generate new set of inputs. QuickFuzz [22] uses existing Haskell implementations of file-format-handling libraries when producing new test inputs.

3.3 Improving Fuzzing

Similar to TargetFuzz, several prior works enhance one or more aspects of existing fuzzers rather than proposing a new type of fuzzer. For instance, some of the works increase the fuzzing throughput, thereby increasing the chance of generating crashing input. Specifically, Xu et al. [46] propose new operating system primitives to

efficiently handle common operations (e.g, file writing, process spawning) used by fuzzing. Delshadtehrani et al. [11] propose a hardware mechanism that significantly accelerates the throughput of binary fuzzing.

Several works [7, 13, 44, 51] improve the seed scheduling algorithms of available fuzzers. Seed scheduling phase of fuzzing is in charge of choosing a seed for the next fuzz iteration as we detail in Section 2 (i.e., stage **S2** in Algorithm 1). For instance, CollAFL [13] analyzes the control flow graphs of the PUT and prioritizes seeds based on the exercised branches and memory accesses. TortoiseFuzz [44] prioritizes the inputs based on the memory accesses, functions calls, and loops. Hawkeye [7] modifies seed scheduling of AFLGo by prioritizing inputs with lower distances when scheduling a seed for the next iteration. FuzzGuard [51] proposes a deep-learning model that predicts the reachability of seeds to the specific target sites, thereby improving the effectiveness of DGF.

A variety of works [4, 5, 7, 48] change the existing fuzzers to spend more fuzzing time on the inputs that are more likely to lead to the crashing behavior (stage **S3** in Algorithm 1). AFLFast [5] extends AFL [18] with a power scheduling function that relies on a Markov model. AFLFast assigns more energy to those seeds that exercise low-frequency paths. EcoFuzz [48] later replaces the Markov model with a variant of the adversarial multi-armed bandit model for better seed energy assignment. AFLGo [4] later modifies the AFLFast's power scheduling function to direct the fuzzing towards specific target sites and proposes the state-of-the-art directed greybox fuzzer. Hawkeye [7] extends AFLGo by improving its static analysis to compute seed distances more precisely and adapts the mutated seed number based on the proposed distance metric.

4 METHODOLOGY

We propose the design of TargetFuzz, a mechanism that provides a target-specific seed corpus to DGF tools. For each commit that modifies source code of a program, TargetFuzz outputs a different seed corpus consisting of only 'close' seeds that are more likely to lead to the target regions. In the following subsections, we first provide a design overview and then elaborate on the details of each component of TargetFuzz.

4.1 Design Overview

We illustrate the design overview of TargetFuzz in Figure 1. TargetFuzz mainly consists of three components, a Baseline Seed Corpus Generator (BSCG), a Seed Refiner and Accumulator (SRA) and a Target-oriented Seed Selector (TSS). The BSCG is in charge of generating a variety of seeds that cover different code regions of a program. To accomplish this goal, the BSCG performs fuzzing using multiple coverage-based greybox fuzzers by allocating a relatively long time limit and collects all of the 'interesting seeds' (i.e., any seed that increases coverage) from each fuzzer's output queue in one place. The fuzzer-generated interesting seeds comprise a large seed pool that is used in a later stage of TargetFuzz when selecting seeds for a commit-specific corpus. We refer to this large seed pool as *baseline seed corpus* and the elements of these corpus as *baseline seeds*. TargetFuzz runs the BSCG for two different scenarios. The first scenario is when TargetFuzz is initially integrated into the testing framework of a software project. The second scenario is when the software project undergoes substantial changes. This is an occasional scenario in a software project as the maintenance of real-world projects mostly involves small size commits that are submitted with high frequency.

Once the baseline seed corpus is generated, TargetFuzz waits until a developer submits a new commit that modifies the program source. After the submission of the commit, TargetFuzz runs the SRA component to identify crashing inputs and to eliminate duplicate seeds. Specifically, the SRA first checks if any of the available seeds in the baseline seed corpus triggers a bug introduced by a new commit. Next, the SRA refines the baseline seed corpus since different coverage-based fuzzers used in the BSCG can generate the same and/or similar inputs. In particular, the SRA retains only the smallest set of seeds that result in the same code coverage. The seeds after the refining process are used to form the *accumulated seed corpus*.

The TSS selects a subset of seeds from the accumulated seed corpus and outputs a target-specific seed corpus which we refer to as *DART corpus*. While selecting seeds of the DART corpus, the TSS takes the seed distance into account to select only those seeds that are 'close' to the modified code region. The TSS takes the modified program source to compute seed distances. TargetFuzz equips the directed greybox fuzzer with the DART corpus, sets the modified lines of code as target sites, and starts the fuzzing session. Once the fuzzing session is over, TargetFuzz first checks if a new bug is detected. Next, the fuzzer-generated 'interesting' new seeds (i.e., seeds that increased coverage during fuzzing) are provided to to the SRA, which augments the accumulated seed corpus. When the next commit is submitted, the SRA first refines the augmented accumulated seed corpus to eliminate redundant seeds obtained from the previous fuzzing session. Note that the SRA uses the accumulated seed corpus for next commits (instead of the baseline seed corpus) until the BSCG needs to generate a new baseline seed corpus because of a major program change.

4.2 Baseline Seed Corpus Generator (BSCG)

The BSCG creates a large seed pool consisting of a variety of seeds that cover different code regions of the program. To provide a commit-specific DART corpus to a DGF tool, TargetFuzz applies its seed selection strategy on the accumulated seed corpus mostly consisting of baseline seeds along with a limited number of new seeds generated by DGF. This is because TargetFuzz allocates very limited time for fuzzing small-size commits, and so it is unlikely to achieve a promising code coverage with new seeds (i.e., the output of Directed Greybox Fuzzer in Figure 1) coming from DGF sessions. If the baseline seeds cover only a certain region of the program or a limited number of easy-to-reach program regions, the DART corpus will include 'further away' seeds for most of the commits and fail to guide DGF towards the target code regions. To ensure that the DART corpus includes a set of 'close' seeds that are related to program regions modified by the recent commit, it is essential to collect a wide range of baseline seeds that exercise different regions of the program. To this end, TargetFuzz runs multiple coverage-based fuzzers with different characteristics since none of the prior fuzzers manifest clear superiority over the others [24, 29]. Additionally, to improve the code coverage, TargetFuzz allocates a relatively longer

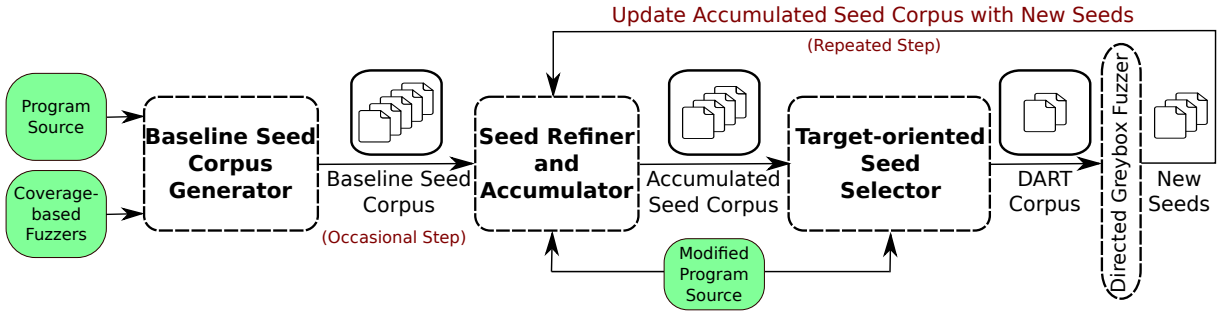


Figure 1: TargetFuzz Design: The inputs of TargetFuzz are color-coded with green. The Baseline Seed Corpus Generator produces a wide range of seeds using CGF tools. Seed Refiner and Accumulator eliminates the baseline seeds with similar characteristics and accumulates incoming seeds from the fuzzing sessions of submitted commits. Target-oriented Seed Selector selects commit-specific seeds from the accumulated seed corpus and outputs a DART corpus.

fuzzing time limit for this step compared to DGF sessions. However, it is important to note that this step is occasional as it is performed only after a major software change and the time cost of this step is negligible in the long term given the high frequency of small-size commits.

4.3 Seed Refiner and Accumulator (SRA)

The SRA initially checks if the modified program contains a bug that can be detected with any of the seeds in the accumulated seed corpus. If a bug is detected with a seed that resides in the accumulated seed corpus, the developers can fix the bug with a follow-up commit before running any fuzzing sessions. In case the developers ascribe low priority to a detected bug, it is essential to ensure that the bug is not a 'fuzz blocker' that limits the effectiveness of fuzzing [12].

As a next step, the SRA eliminates seeds that result in the same branch coverage since using the same type of seeds leads to wasted fuzzing effort [39]. The SRA uses two practices that are widely used by existing corpus minimization tools [31, 49]. First, the SRA eliminates seeds that achieve the same branch coverage. The SRA chooses only one seed from each set of seeds that produces the same branch coverage. Second, we choose the seed with the minimum file size among all the seeds that result in the same branch coverage. As discussed by prior works [33, 46, 49], here the rationale is to reduce the search space for mutation while minimizing I/O requests.

4.4 Target-oriented Seed Selector (TSS)

As detailed before, the time limits of DGF fuzzing sessions are significantly limited (usually less than 24 hours). TargetFuzz constructs a seed corpus consisting of "closer" inputs that guide DGF towards the modified code regions, thereby reaching target sites under a limited fuzzing time. To steer the execution towards target sites (e.g., modified lines of code after a commit), the TSS component uses a seed selection strategy that takes the seed distance into account rather than merely code coverage when selecting the seeds for the target-oriented DART corpus. The distance-based DART corpus helps DGF to focus on 'close' seeds that are more likely to lead to the target regions during fuzzing. Additionally, by eliminating 'further away' seeds, the TSS prevents a DGF tool from wasting the limited testing time on fuzzing seeds that are unrelated to the modified region.

The TSS component instruments the program source to generate a DGF-specific binary. The DGF-specific binary is generated by applying necessary compiler passes provided by the directed greybox fuzzer and it includes bookkeeping logic to compute the distance of a provided seed to the given set of target sites. The TSS then executes the instrumented binary with each of the seeds in the accumulated seed corpus to obtain their corresponding seed distance values. Next, based on the computed seed distances, the TSS selects a certain number of 'close' seeds (with smallest seed distances) from the accumulated seed corpus and assembles these seeds in the DART corpus. Here, one important design aspect of TargetFuzz is the process of determining DART corpus size. At one extreme, the TSS can select the single closest seed to include the most related seed to the target region in DART corpus. At another extreme, the TSS can select all the seeds to increase the variety in DART corpus. Overall, there is a trade-off between the closeness (~quality) and variety (~quantity) of seeds. In Section 5, we explain the details of our implementation choice that tackles this problem.

A DGF tool uses the DART corpus for fuzzing the modified program. After the fuzzing time limit is over, the new 'interesting' seeds obtained from this particular fuzzing session are added to the accumulated seed corpus. The new seeds can potentially cover a program region that is not covered by any of the seeds in the accumulated seed corpus, thereby increasing diversity of the accumulated seed corpus. Whenever a new commit arrives, the SRA processes the augmented accumulated seed corpus and updates it by refining redundant seeds.

5 IMPLEMENTATION

In this section, we provide the implementation details of TargetFuzz for each component presented in Figure 1.

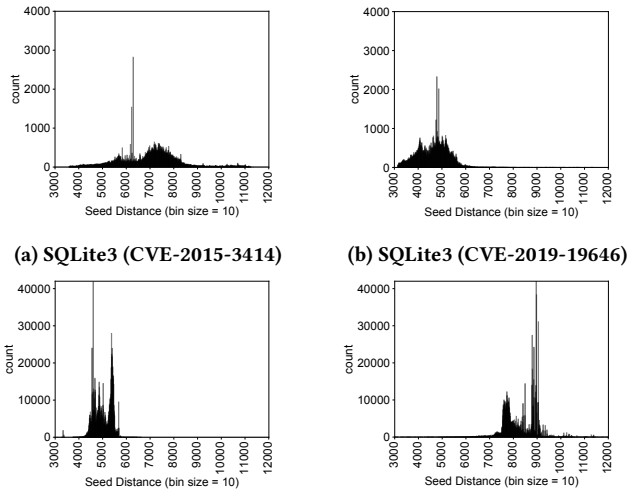
BSCG. To generate the baseline seed corpus, we used four different coverage-based fuzzers deployed as part of a fuzzing environment, namely Magma [24]. The coverage-based fuzzers are AFL [18], Angora [8], MOpt-AFL [32], and honggfuzz [21]. For each software that we tested, we run separate fuzzing sessions using each of the fuzzers for a certain time limit (detailed in Section 6). Once the

fuzzing sessions are over, we collect all the coverage-increasing seeds residing in the output queues of each fuzzer and assemble them in the baseline seed corpus.

SRA. To increase the chance of detecting bugs in the program, we compiled the program source using LLVM passes that insert the sanitizer checks such as Address Sanitizer. The SRA also eliminates any seeds which made the application hang or require more than a fixed-memory limit determined by the fuzzers (which is 100MB in our case). To eliminate the duplicate seeds, the SRA utilizes the two previously discussed practices that are implemented as part of `afl-cmin` [49], an existing seed minimization tool. The SRA uses `afl-showmap` to extract the edges exercised by each seed. Note that our seed deduplication uses `afl-cmin`'s logarithmic branch hit count mode (i.e., no `-e`) when measuring branch coverage.

TSS. We implement TargetFuzz on top of AFLGo [4]. We use AFLGo because it is one of the state-of-the-art directed greybox fuzzers and it is open-sourced. The TSS compiles the target program by using AFLGo's LLVM passes. These LLVM passes modify program source to include the book-keeping logic and to compute the distance of a provided seed to the given set of target sites by analyzing control-flow graphs and call graphs of the program. We leave the details of these passes in terms of the performed static analysis to the original AFLGo paper [4] and the github repo of AFLGo [2]. The TSS also sets the target sites as the changed code lines after a commit. After the generation of an instrumented binary, we perform a dry-run with AFLGo to compute the distance values of baseline seeds and use the same binary for fuzzing.

As described in Section 4, there is a trade-off between the quality and the quantity of seeds that comprise DART corpus. Here, we explain how we tackle this trade-off by providing the implementation details of the TSS. Specifically, we explain how TargetFuzz determines the total number of seeds that construct the DART corpus (i.e., the size of DART corpus). Before we explain our approach, we deem it useful to present the seed distance distributions for two programs (SQLite3 and Libxml2) as histograms in Figure 2. We choose two different target sites per program where each target site refers to code lines that introduce a known CVE. The seed distance distributions demonstrate several key points that we take into account in our approach: **(K1)** A seed distance distribution does not necessarily conform to a common probability distribution (e.g., normal distribution) as in several examples provided in Figure 2. Therefore, it is not feasible to forecast a seed distance value by using metrics that define a specific probability distribution such as mean and standard deviation for normal distribution. **(K2)** Seed distance distributions that belong to two different programs can significantly vary. As presented in Figure 2a and Figure 2c, SQLite3 and Libxml2 libraries show different characteristics in their seed distance distributions. Therefore, the selection strategy should ensure that it works smoothly for different types of programs. **(K3)** Seed distance distributions that belong to the same program can vary significantly when different target sites are chosen. As shown in Figure 2a and Figure 2b, even the same program (i.e., SQLite3) can present varying seed distance distributions for two different code targets. This point shows that it could be misleading to rely on program characteristic when selecting seeds. **(K4)** Seed distance distributions contain outliers as we observe in several programs. Selecting seeds from outliers can potentially misguide DGF.



(a) SQLite3 (CVE-2015-3414) (b) SQLite3 (CVE-2019-19646)

(c) Libxml2 (CVE-2015-8317) (d) Libxml2 (CVE-2017-9047)

Figure 2: The seed distance distributions for different code patches.

Based on the provided seed distance distributions, there is no clear seed distance cut-off value that applies to all programs and target sites. TargetFuzz proposes to use a percentile-based seed selection strategy that considers the above discussed points. TargetFuzz first calculates the k^{th} percentile of a set of seed distance values. Next, the calculated percentile value is used as a threshold when selecting the seeds from the baseline seed corpus. The TSS determines all the seeds that have a lower distance than the threshold and assembles those seeds in the DART corpus. To show the trade off between the quantity and quality of seeds, we evaluate TargetFuzz using different percentile values. We provide the details in Section 6.4. Our threshold method uses percentiles for their following advantages: **(1)** Percentiles are versatile. They do not depend on a specific probability density function and can be calculated regardless of the probability distribution [34]. As pointed out by **(K1)**, **(K2)**, and **(K3)**, this property is essential to determine a seed distance threshold for different distributions that belong to different programs and targets. **(2)** Percentiles are resistant to outliers in the data set [6], which occur frequently in seed distance data, as we pointed out in **(K4)**.

6 EVALUATION

In this section, we evaluate the effectiveness and efficiency of TargetFuzz using real-world programs. We first test TargetFuzz using the fuzzing benchmark Magma on a number of real bugs to assess its impact on the bug-finding capability of a DGF tool. Specifically, we want to assess if a corpus tailored for DGF (i.e., DART corpus provided by TargetFuzz) outperforms CGF-based corpora in terms of bug-finding capability. Second, for the same set of real bugs, we measure the performance boost of a DGF tool when equipped with our distance-based DART corpus rather than CGF-based corpora. This aspect of evaluation is important to show that AFLGo can successfully expose bugs even under limited fuzzing time as real-world

applications have limited testing time due to the high-frequency of commits. Last, we evaluate TargetFuzz on several commit sequences obtained from real-world git repositories to demonstrate that TargetFuzz can successfully be deployed in real-world continuous integration (CI) systems. We analyze the achieved code coverage in target code regions (i.e., lines of code modified by the commit) for each commit to understand if TargetFuzz successfully guides DGF to the target regions. Overall, our evaluation aims to answer the following questions:

RQ1: How effectively does TargetFuzz aid a DGF tool in detecting bugs (Section 6.4)? Does DART corpus present a clear advantage over CGF-based corpora?
RQ2: Does TargetFuzz improve the bug-finding speed of a DGF tool (Section 6.5)?
RQ3: Can TargetFuzz effectively guide DGF tools to the target code regions (Section 6.6)?

6.1 Evaluation Dataset

To assess TargetFuzz in terms of its effectiveness in bug detection (**RQ1**) and in terms of its efficiency in bug-finding speed (**RQ2**), we use Magma [24], a fuzzing benchmark suite that includes a variety of real bugs as part of seven different real-world programs. We choose Magma in our evaluation for two main reasons. First, Magma is a ground-truth fuzzing benchmark that enables us to provide an accurate quantitative evaluation. We can easily compare our new mechanism (TargetFuzz) with prior works based on the known bugs that reside in the provided program by Magma. Second, Magma follows an approach called *forward-porting* that is well-suited for the evaluation of TargetFuzz. As detailed before, the real-world projects frequently rely on small size changes. Similarly, Magma provides a separate patch file for each bug where each patch modifies a certain region of the program with a limited number of code line changes. As a reference point, the average patch size of all the Magma bugs that we used in our evaluation is 15 lines of code which is in the same order of the commit size of the real-world projects (i.e., 34.8 lines as detailed in Section 1). By applying these small-size Magma patches (one patch at a time) on a complex real-world program, we imitate a scenario where a recent commit modifies a certain region of the program. For fuzzing, we set the target sites as the lines modified by the imitated commit.

In our experiments, we test all the seven libraries integrated as part of Magma; libpng, LibTIFF, Libxml2, Poppler, SQLite3, OpenSSL, and PHP. It is not feasible to directly fuzz a library, and so the common practice to overcome this is to prepare driver programs. For each library, Magma provides one or more driver programs (in total 26 drivers for 7 libraries presented in Table 3) that call functions in the library. Magma does not provide the name of the driver program(s) that triggers a certain bug, and so one needs to fuzz each library with all the available driver programs. Evaluating TargetFuzz (and related works) on all the bugs available in Magma (in total 118) using all the provided drivers requires $\sim 1M$ CPU hours, which is infeasible for our resources. Therefore, we rely on the proof of concepts (PoC) provided by Magma [23] to understand the driver program of each bug (in total 58 PoCs). For each library, we pick one driver program (highlighted in Table 3) that exposes the maximum number of bugs. In total, this adds up to 34 bugs. The selected bugs have different bug classes such as integer overflow, heap buffer overflow, uninitialized memory access, etc.

Table 3: Driver programs of libraries provided by Magma.

Library	Drivers
libpng	**read_fuzzer**, readpng
LibTIFF	read_rgba_fuzzer, **tiffcp**
Libxml2	**xml_reader_for_file_fuzz**, xmllint
Poppler	pdf_fuzzer, **pdfimages**, pdftoppm
OpenSSL	asn1, asn1parse, bignum, bndiv, client, cms, conf, crl, ct, **server**, x509
SQLite3	**sqlite3_fuzz**
PHP	**exif**, json, parser, unserialize

6.2 Infrastructure and Settings

As described in Section 4, TargetFuzz has a seed collection phase (i.e., baseline seed corpus generation) that creates a large seed pool with a variety of fuzzer-generated baseline seeds. To generate the baseline seeds, we fuzzed the bug-free version[3] of each library using its application driver highlighted in Table 3. We run four instances of each of the coverage-based fuzzers (i.e., AFL, Angora, MOPT-AFL, honggfuzz) for each library where each instance uses one core and has a time limit of 168 hours (7 days). In total, collecting baseline seeds for 7 application drivers took 18816 CPU hours ($168hours \times 4instances \times 4fuzzers \times 7drivers$).

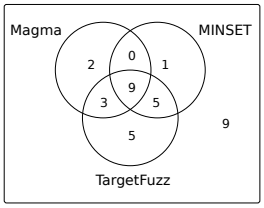
The collected baseline seeds were used to test ground-truth bugs in Magma. Specifically, to test each bug, we first applied its corresponding patch to the clean library version. Next, AFLGo was equipped with a patch-specific DART corpus to fuzz the patched library for 24 hours. To evaluate the impact of the quantity and quality of seeds that comprise DART corpus, we used 0.1^{st},1^{st}, and 10^{th} percentiles to output three different DART corpora which we refer to as DART-0.1^{st}, DART-1^{st}, and DART-10^{th}, respectively. We used a logarithmic scale to show the impact of percentiles more explicitly. Due to the probabilistic nature of fuzzing, we run 20 fuzzing instances for each patched library and report the geometric mean of Time-to-Exposures (TTE). TTE is the total duration from the beginning of the fuzzing session until the bug is triggered. A run that did not reproduce the vulnerability within 24 hours received a TTE of 24 hours. The dedicated CPU resources for each fuzzing instance was one CPU core and 1GB of memory. AFLGo was configured with the parameters used in the original paper (`-z exp -c 4h`). We used the most recent commit of the AFLGo repo [2] at the time we started conducting experiments (commit `e27a908`). All the experiments were conducted in virtual machines started on server nodes with Intel® Xeon® Gold 6132 CPUs and Ubuntu 18.04 LTS as the operating system.

We compared DART corpus with two different CGF-based seed corpora. The first corpus is provided by Magma and consists of generic seeds well-suited for CGF. Specifically, Magma sources these seed from the library repositories, OSS-Fuzz [20] and the AFL repository [18]. As shown in Table 1, Magma corpus is similar to the corpus used in the original AFLGo experiments [4]. As part of

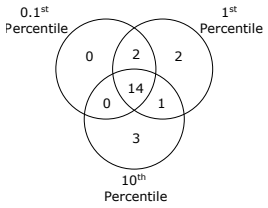
[3]Throughout the paper, we refer to a library version as bug-free if it does not include any of the ground-truth bugs provided by Magma.

Table 4: Corpora size.

Library	Baseline Seed #	DART Seed #			MINSET Seed #
		0.1^{st}	1^{st}	10^{th}	
libpng	28,048	51	238	2425	172
LibTIFF	127,425	48	315	782	76
Libxml2	1,095,819	1583	9663	99503	496
Poppler	164,806	113	221	944	133
SQLite3	119,323	234	1659	11938	639
OpenSSL	39,949	133	336	1999	187
PHP	21,556	17	167	1668	77



(a) The quantity of found bugs for different corpora.



(b) The impact of percentiles on bug detection.

Figure 3: Unique bug detection.

our experiments, we use Magma corpus rather than AFLGo corpus since Magma corpus is constructed more recently and therefore contains more up-to-date seeds. Since developers from both OSS-Fuzz and the tested libraries improve their seed corpora over time, using an outdated seed corpus (e.g., AFLGo corpus) could bring unfair performance benefits in our experiments.

As shown by a recent work [25], there is no clear superiority of any available CGF-based corpus minimization tools (i.e., `OPTIMIN`, `afl-cmin`, and `MINSET`) over the others. We assembled the seeds in the second corpus by using one of these tools, specifically `MINSET` [39], which showed promising results with CGF tools. To evaluate `MINSET`, we applied the greedy algorithm (detailed in Algorithm 2) to the accumulated seed corpus[4] in Figure 1 by removing the TSS component. In total, we spent 81600 CPU hours ($24hours \times 20instances \times 5corpora \times 34bugs$) to compare the effectiveness of distance-based DART corpora with CGF-based seed corpora.

6.3 Seed Corpus Size

This section provides the details related to seed corpora sizes. In the second column of Table 4, we report the baseline number that CGF-based fuzzers generated by fuzzing bug-free version of each library in Magma. Additionally, we calculate the corpora sizes of DART corpus for three different percentile values (column 3-5) and `MINSET` corpus (column 6) after applying their corresponding seed selection algorithms to the baseline seed corpus. As shown in Table 4, the libraries that we tested with TargetFuzz resulted in different sizes of baseline seed corpora. This observation is important to demonstrate that our distance-based seed selection strategy is effective not only for certain sizes of seed pools. In fact, the distance-based seed selection strategy worked effectively for baseline seed corpora with different scales. As an example, TargetFuzz detected almost all the ground-truth bugs (3 out of 3 in OpenSSL and 4 out of 5 in Libxml2) in two libraries which differ by orders of magnitude in their baseline seed number (i.e., OpenSSL(~40K) and Libxml2 (~1M)). As we detail in Section 6.2, we allocated the same fuzzing time limit for all the libraries to create each baseline seed corpus. Therefore, there is a difference in the number of collected baseline seed result based on the different characteristics of libraries.

[4]It is evident that seeds with high seed distance values are less likely to lead to the target sites. Therefore, for a more fair comparison, we eliminate any seed with a distance value higher than the geometric mean of the population from baseline seed corpus.

Table 4 includes the corpus minimization results for `MINSET`, DART-0.1^{st}, DART-1^{st}, and DART-10^{th} corpora. Due to varying seed distance distributions of different programs (see Section 5), the same percentile value can result in different DART seed number. For instance, libpng and PHP libraries have different DART seed number for the 0.1^{st} percentile although they have similar baseline seed number. Other than Libxml2 library, the size of `MINSET` corpus is always between the size of DART-0.1^{st} and the size of DART-1^{st}.

Additionally, DART-10^{st} corpus contains at least an order of magnitude more seeds than `MINSET` corpus for all the libraries. As detailed in next sections and shown by prior work [25], the corpus size by itself is not the ultimate metric to evaluate a seed selection strategy. Different seed selection strategies can result in different corpus size and the increase in corpus size does not necessarily increase the effectiveness of a fuzzer as shown by our work and also by Herrera et. al [25]. However, it still helps us to interpret certain aspects of our evaluation results, especially when interpreting why some of the corpora achieve/fail to discover certain bugs.

6.4 Unique Bug Detection

Overall results. As discussed by several works [27, 33], the bug-finding capability of a fuzzer should be the ultimate metric for the evaluation. To this end, we evaluate the bug-finding capability of AFLGo when equipped with different types of corpora; DART corpus, Magma corpus, and `MINSET` corpus. Specifically, we equip AFLGo with each type of corpora separately, fuzz each buggy program version, and check if at least one of the twenty fuzzing sessions triggers the bug.

Our main goal here is to show that a DGF-based corpus outperforms a CGF-based corpus when used with a DGF tool. We present our results as a Venn diagram in Figure 3a. Unfortunately, AFLGo could not trigger 9 out of 34 bugs with any of the seed corpora[5]. DART corpora (i.e., DART-0.1^{st}, DART-1^{st}, and DART-10^{th}) triggered 22 out of 25 remaining bugs in total where we provide the details of the sensitivity analysis later in this section. Magma and `MINSET`-based corpus could trigger 14 and 15 bugs, respectively. In total, DART corpora discovered 5 unique bugs while Magma and `MINSET`-based corpora discovered 2 and 1 unique bugs, respectively. Overall, our experimental results demonstrate that a seed corpus created with a CGF-based seed selection strategy (i.e., `MINSET`) can

[5]The library and id of each bug are as follows; Libxml2 (`AAH035`), Poppler (`AAH049`, `JCH212`), SQLite3 (`JCH216`, `JCH223`, `JCH226`, `JCH227`), and PHP (`MAE008`, `MAE014`).

Table 5: Speedups achieved by Magma, MINSET, and DART corpus. We provide TTE (in seconds) for each bug when tested with AFLGo using different corpora. A maximum of 86400 sec is assigned to a bug if it is not found within the reserved time (24h).

Library	Bug No	Magma	MINSET	DART			Speedup			
				0.1^{st}	1^{st}	10^{th}	MINSET	0.1^{st}	1^{st}	10^{th}
libpng	AAH001	86400	86400	84804	86079	86400	1.00	1.02	1.00	1.00
	AAH007	86400	277	86400	15	69	312.20	1.00	5,760	1,261
	AAH008	59910	86400	68168	62491	86400	0.69	0.88	0.96	0.69
LibTIFF	AAH010	86400	79564	86400	86400	86400	1.09	1.00	1.00	1.00
	AAH014	37835	10	10	10	10	3,784	3,784	3,784	3,784
	AAH017	52326	74227	49053	42116	42116	0.70	1.07	1.24	1.24
	AAH020	57084	21533	34219	13799	16170	2.65	1.67	4.14	3.53
	AAH022	81682	818	81649	13795	57948	99.89	1.00	5.92	1.41
Libxml2	AAH024	86400	86400	1381	33887	86400	1.00	62.55	2.55	1.00
	AAH025	86400	86400	86400	82104	86400	1.00	1.00	1.05	1.00
	AAH026	86400	86400	86400	18115	74342	1.00	1.00	4.77	1.16
	AAH032	86400	82128	34334	4674	35761	1.05	2.52	18.49	2.42
Poppler	AAH043	86400	84461	51594	51594	71300	1.02	1.67	1.67	1.21
	AAH045	86400	86400	86400	86400	85317	1.00	1.00	1.00	1.01
	AAH050	86400	86400	647	661	86400	1.00	133.49	130.76	1.00
	JCH201	44299	86400	86400	86400	86400	0.51	0.51	0.51	0.51
	JCH209	86400	6028	6219	6321	5963	14.33	13.89	13.67	14.49
	JCH210	86400	17533	21225	20075	82833	4.93	4.07	4.30	1.04
SQLite3	JCH215	53302	47081	26327	26327	26327	1.13	2.02	2.02	2.02
	JCH228	21805	86400	86400	86400	86400	0.25	0.25	0.25	0.25
	JCH232	79222	86072	86400	86400	82545	0.92	0.92	0.92	0.96
OpenSSL	AAH055	15721	86400	236	1777	75112	0.18	66.69	8.85	0.21
	AAH056	610	95	367	367	1315	6.44	1.67	1.67	0.46
	MAE115	80898	8499	86400	86400	26476	9.52	0.94	0.94	3.06
PHP	MAE016	800	15	3908	3778	3541	53.33	0.20	0.21	0.23
Geo. mean	-	45225	14968	16744	11220	23035	3.02	2.70	4.03	1.96

perform poorly when used with a DGF tool. Moreover, a seed corpus well-suited for CGF (i.e., Magma corpus) can fail to trigger bugs with a directed greybox fuzzer. By taking the seed distance metric into account in the seed selection strategy, the bug detection capability of DGF improves as demonstrated with TargetFuzz.

Sensitivity analysis. As discussed in Section 5, TargetFuzz uses percentiles to determine the size of DART corpus. By choosing three different percentile values with different orders of magnitude, we evaluate how the quality and quantity of seeds in DART corpus impact the bug-detection capability of a DGF tool. We demonstrate the impact of the chosen percentile value on the effectiveness of bug detection as a Venn diagram in Figure 3b. Our results demonstrate that the size of DART corpus clearly affects the bug-finding capability of DGF. Specifically, DART-0.1^{st} corpus detected least number of bugs although it is constructed with the closest seeds to the target sites. In fact, DART-1^{st} corpus triggered all 16 bugs that DART-0.1^{st} corpus triggered and three more bugs that were not triggered by 0.1^{st} percentile corpus. The main reason behind the ineffectiveness of DART-0.1^{st} corpus is the limited variety of seeds compared to DART-1^{st} and DART-10^{th}. A clear example is `AAH007` bug where a simple bit-flip operation of a baseline seed easily triggers the bug. Since DART-0.1^{st} does not include this seed as part of the corpus (i.e., low variety), it failed to trigger this bug after 24 hours of fuzzing. In fact, five out of six bugs, which were not triggered with DART-0.1^{st} corpus, were discovered with less than two hours of fuzzing after a set of mutations on one of the seeds that resides in either a DART-1^{st} corpus or a DART-10^{th} corpus. While DART-1^{st} and DART-10^{th} corpora did not show clear superiority (4 and 3 unique bugs, respectively) over each other in terms of bug detection capability, they differ significantly in their time-to-exposure as detailed in Section 6.5.

6.5 TTE Speedups

We also demonstrate that TargetFuzz helps DGF to reduce the time-to-exposure of bugs, which is important especially when the project has limited time for fuzzing. To show the efficiency of the DART corpus over the Magma corpus and the `MINSET` corpus, we report the TTE of bugs (in seconds) and the speedups in Table 5. When calculating the speedups, we use the results obtained with Magma corpus as a baseline since it performed the worst. In summary, `MINSET`, DART-0.1^{st}, DART-1^{st}, and DART-10^{th} resulted in 3.02×, 2.7×, 4.03×, 1.96× speedup over Magma corpus, respectively.

Table 5 presents that `MINSET` corpus (4^{th} column) outperformed Magma corpus (3^{rd} column). Within the first 10 hours (36000 seconds) of fuzzing, `MINSET` corpus triggered several bugs (e.g., `AAH007`, `AAH014`, `AAH022`) which could not be triggered by Magma corpus.

Evidently, the efficiency of MINSET corpus over Magma corpus mainly comes from the usage of baseline seeds. As an example, MINSET corpus easily triggered AAH014 after AFLGo applied a single bit-flip on a seed selected from the baseline seed corpus.

The performance benefit of DART corpus over MINSET corpus depends on the percentile value that TargetFuzz uses. DART-1^{st} corpus leads to best results among the three percentile-based corpora and it outperforms MINSET corpus in bug-finding speed averaged for 25 bugs. Compared to MINSET, it discovered several bugs significantly faster (e.g., AAH032, AAH017, and JCH215) and was able to discover more bugs within the time limit (e.g., AAH050, and AAH055). These examples clearly present the benefit of using distance metric in the seed selection strategy rather than only relying on code coverage. However, the important observation here is that DART-0.1^{st} and DART-10^{th} are less efficient than MINSET although the seeds in those two corpora are also selected using seed distance metric.

The advantage of DART-1^{st} corpus over DART-0.1^{st} corpus is the variety and quantity of seeds. For instance, for some of the seed distributions, we observe that the distance values in the 1^{st} percentile are very close to each other. In this scenario, choosing 0.1^{st} over 1^{st} percentile prevents fuzzer from using a high number of 'close' seeds as part of the seed corpus and therefore it is less likely for the fuzzer to explore new paths with mutations. The scenarios discussed above for 0.1^{st} and 1^{st} percentiles comparison are valid for 1^{st} and 10^{th} percentiles as well. As we demonstrate in Figure 3b, DART-10^{th} corpus identified three unique bugs (i.e., AAH045, JCH232, MAE115) that could not be triggered by DART-0.1^{st} and DART-1^{st} corpus. While DART-0.1^{st} and DART-1^{st} corpus could not trigger these bugs due to the limited variety of seeds in their corpora, DART-10^{th} triggered it by a sequence of mutations on a seed with a distance value between 1^{st} and 10^{th} percentile.

Overall, our experimental results show that there needs to be a balance between the quantity and the quality of seeds. While a couple of bugs could be identified by only DART-10^{th}, the effectiveness of DART-1^{th} corpus is the highest in terms of bug-finding capability. As the number of seeds that construct the seed corpus increases, a DGF tool allocates less time for each seed. Therefore, the fuzzer performs a lower number of mutations on each seed (especially for DART-10^{th}). As an example from Table 4, AFLGo starts with 99500 and 9663 seeds (on average) for Libxml2 library when 10^{th} and 1^{st} percentiles are chosen. The 10× difference in corpus size significantly reduces the total time that the fuzzer spends on each seed including seeds that are likely to lead to the buggy region with mutations. This increases TTE significantly for several Libxml2 bugs including AAH024, AAH026, AAH032. Similar scenario is valid for bugs residing in other libraries such as AAH050 from Poppler, AAH055 from OpenSSL.

6.6 Continuous Fuzzing

In addition to the Magma benchmark, we tested TargetFuzz on several commit sequences obtained from the real-world project repositories. The goal of this experiment is to show that TargetFuzz can successfully be deployed as part of CI systems. For instance, Google's OSS-fuzz has a service that builds a project using its source code at a particular commit and subsequently runs a fuzzing session. Currently, OSS-fuzz includes only coverage-based fuzzers [15] (afl, honggfuzz, libFuzzer) and provides a corpus minimization tool tailored for coverage-based fuzzers [19]. TargetFuzz outputs DART corpus tailored for DGF; therefore, it can be potentially used as part of OSS-fuzz if a directed greybox fuzzer like AFLGo is actively used and maintained in the OSS-fuzz infrastructure.

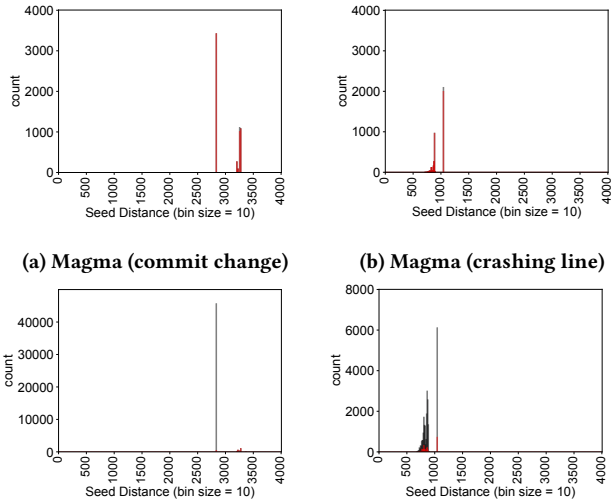
Table 6: Total line coverage percentage achieved by fuzzing SQLite3, LibTIFF, and libpng over a commit sequence.

Library	Commit	Line Coverage (%)		Total Line #
		Magma	DART	
SQLite3	14c4d42	82.14	100	28
	c00727a	0	100	21
	bf7f3a0	100	100	10
	542812	0	80	10
	7cc73b3	0	100	5
	be12083	10	100	10
Average	-	32.02	96.67	13.16
LibTIFF	4ecf751	0	0	10
	de7617a	93	100	15
	f0f68dc	66.67	66.67	3
	120aa39	0	0	54
	86a8232	0	0	2
	1c7e305	100	100	2
Average	-	43.33	44.44	14.3
libpng	a37d483	0	0	11
	3796518	100	100	1
	c4bd411	0	100	1
	eb67672	0	100	2
Average	-	25	75	2.1

Since there are no available ground-truth for project repositories, we rely on code coverage. Specifically, we use the line coverage as the metric since the lines of code need to be executed in order to find a bug. The details of our evaluation data set are provided in Table 6. We used the latest commits from the master branch of three different libraries SQLite3, LibTIFF, libpng at the time we started the experiments. The commit numbers are provided in the second column. We preferred these libraries because of their lower compilation times when generating AFLGo-specific binaries. As a reference point, it takes ~6 hours to compile OpenSSL and PHP while it is around 20 minutes for LibTIFF. The commit numbers are listed from newest (top) to oldest (bottom) for each library (e.g., 14c4d42 and be12083 are the newest and the oldest commits for SQLite3, respectively). The time limit is determined for each repository based on the commit frequency to that repository. Specifically, we dedicated 12, 24, and 24 hours to SQLite3, LibTIFF, and libpng libraries, respectively. After fuzzing each commit with a commit-specific DART corpus, we collected the all the seeds that increased coverage in the fuzzing sessions and combined them with the available accumulator seed corpus as described in Section 4. For this experiment, we used Magma corpus as a comparison point to DART-1^{st} corpus. As described in Section 6.2, Magma corpus contains a subset of seeds sourced from the actual repositories. Therefore, our evaluation aims to compare the effectiveness of AFLGo when equipped with a seed corpus sourced from the actual repository and DART corpus.

The achieved line coverage by Magma and DART corpora are provided under the third and fourth columns of Table 6, respectively. The results present the advantage of using DGF-based DART corpus over a generic CGF-based corpus. For SQLite3 library, DART corpus achieved 3× more line coverage over Magma corpus (32% and 96%). For several commits, Magma corpus failed to reach the target code lines. Similar observations apply to libpng library as well, where DART corpus achieved 3× more line coverage over Magma corpus (25% and 75%). The line coverage results for these two libraries clearly signify the contribution of TargetFuzz when testing a certain program region under a limited fuzzing time.

The coverage results of DART corpus for LibTIFF do not present a clear advantage over Magma corpus. However, interestingly, AFLGo detected two unique bugs (stack overflow and heap-based buffer overflow) when fuzzing `1c7e305`, `120aa39`, and `4ecf751` commits only with DART corpus. We confirmed that the bugs reside in the most recent commit of the repository as well. We reported the stack overflow bug to the developers and also observed that the heap-based buffer overflow bug was an old bug reported by another group in the past. Our further analysis showed that none of the bugs reside in the modified lines of the code by the commits. To realize why using DART corpus resulted in bug detection while using Magma corpus failed, we plotted the seed distributions in Figure 4. Specifically, we provide four different seed distribution histograms that are generated using seeds collected from AFLGo's fuzzing sessions (including seeds that form initial seed corpus) for Magma corpus and DART corpus. Figure 4a and Figure 4c present seed distributions when we set the modified lines by `4ecf751` commit as targets. Figure 4b and Figure 4d show seed distributions when the buggy line that caused the stack overflow is set as target. In each figure, we highlighted seeds generated during fuzzing with red. The histograms provide several takeaways. First, for both Magma and DART corpus, the collected seeds are significantly closer to the crashing line which can be observed from the seed distance values (e.g., seed distances in Figure 4c and Figure 4d for DART corpus). Therefore, some of the seeds reached buggy line of the code during fuzzing with DART corpus and eventually triggered the bug. Second, when using Magma corpus as an initial seed corpus, AFLGo generated a variety of seeds to cover certain program regions which were already covered by some of the seeds in DART corpus. Indeed, in Figure 4d, we see a subset of seeds generated during fuzzing session (highlighted with red) accumulating around 800 distance. While AFLGo spends the limited fuzzing-time to generate seeds around ~800 when equipped with Magma corpus (see Figure 4b), DART corpus provides seeds with ~800 seed distance as part of the initial seed corpus. Therefore, AFLGo is served with seeds closer to the buggy line by DART corpus when the fuzzing session starts and eventually performing a sequence of mutations on one of the seeds with seed distance value ~800 resulted in discovering the bug.



(a) Magma (commit change) (b) Magma (crashing line)

(c) DART (commit change) (d) DART (crashing line)

Figure 4: The seed distance distributions for Magma and DART corpus when setting targets as modified lines by `4ecf751f` commit and crashing line of buffer overflow bug.

7 DISCUSSION

Seed trimming. Seed trimming approaches such as `afl-tmin` [50] and MoonShine [37] attempt to transform a seed into another version that is smaller in size yet achieves the same code coverage as the original seed. Seed trimming is useful since smaller seeds consume less I/O (thus having higher throughput). The seed trimming approaches are orthogonal to our work and can be readily applied within TargetFuzz, too. Specifically, we can leverage seed trimming to reduce seed sizes in the baseline seed corpus and DART corpus.

Baseline seed corpus. As the software evolves with new features, the coverage of baseline seeds can potentially change. When the baseline seeds do not cover the project well (i.e., the code coverage decreases), it is necessary to update the baseline seed corpus. The update frequency of the baseline seed corpus is highly project dependent since different projects have different characteristics. For instance, in recent years, the libpng developers submit very limited number of commits where each commit modifies a few lines of code. However, PHP developers are more active (i.e., higher commit frequency where some of the commits modify a large number of lines). Mechanisms that determine the update frequency of baseline seed corpus are useful, yet out of scope of this paper.

8 CONCLUSION

Existing DGF tools use seed corpora mainly designed for CGF. Providing coverage-based corpora with DGF unnecessarily wastes the fuzzing effort on the unrelated program regions, thereby hindering the bug-finding capability. This work presents TargetFuzz, a mechanism that outputs a seed corpus tailored for DGF. By using the seed distances, TargetFuzz provides DART corpus only consisting of the 'close' seeds to the modified code region. When a DGF tool is equipped with DART corpus, the execution of the DGF tool is steered towards stressing modified code regions, thereby increasing the chance of detecting bugs. Experimental results show that DART corpus improves the bug-finding capability and the speedup in bug detection time of DGF compared to CGF-based corpora.

9 ACKNOWLEDGMENTS

This material is based on research sponsored by BU Hariri Research Incubation Award (#2020-06-005).

REFERENCES

[1] Humberto Abdelnur, Obes Jorge Lucangeli, and Olivier Festor. 2010. *Spectral Fuzzing: Evaluation & Feedback.* Ph.D. Dissertation. INRIA.

[2] aflgo. 2017. AFLGO: Directed Greybox Fuzzing. https://github.com/aflgo/aflgo.

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *IEEE S&P*. 1597–1612.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *ACM CCS*. 2329–2344.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE TSE* 45, 5 (2017), 489–506.

[6] Lutz Bornmann, Loet Leydesdorff, and Rüdiger Mutz. 2013. The use of percentiles and percentile rank classes in the analysis of bibliometric data: Opportunities and limits. *Journal of informetrics* 7, 1 (2013), 158–165.

[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM CCS*. 2095–2108.

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE S&P*. 711–725.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *IEEE S&P*. 1580–1596.

[10] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. Memfuzz: Using memory accesses to guide fuzzing. In *IEEE ICST*. 48–58.

[11] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. Phmon: a programmable hardware monitor and its security use cases. In *USENIX Security*. 807–824.

[12] Firefox. 2021. Fuzzing. https://firefox-source-docs.mozilla.org/tools/fuzzing.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *IEEE S&P*. 679–696.

[14] glennrp. 2018. https://github.com/glennrp/libpng/tree/libpng16/contrib/testpngs.

[15] Google. 2016. OSS-Fuzz. https://github.com/google/oss-fuzz/.

[16] Google. 2020. AFL dictionaries. https://github.com/google/AFL/tree/master/dictionaries.

[17] Google. 2020. AFL test cases. https://github.com/google/AFL/tree/master/testcases.

[18] Google. 2020. American Fuzzy Lop. https://github.com/google/AFL.

[19] Google. 2021. ClusterFuzz. https://google.github.io/clusterfuzz/setting-up-fuzzing/libfuzzer-and-afl/#afl-limitations.

[20] Google. 2021. Continuous Integration. https://google.github.io/oss-fuzz/getting-started/continuous-integration/.

[21] Google. 2021. Honggfuzz. https://github.com/google/honggfuzz.

[22] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. *SIGPLAN Notices* 51, 12 (2016), 13–20.

[23] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma. hexhive. epfl.ch/magma/docs/bugs.html.

[24] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *ACM POMACS* 4, 3 (2020), 1–29.

[25] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *ACM SIGSOFT ISSTA*. 230–243.

[26] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security*. 445–458.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *ACM SIGSAC CCS*. 2123–2138.

[28] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *USENIX Security*.

[29] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, and Peng Cheng. 2021. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security*.

[30] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. 2019. Sequence coverage directed greybox fuzzing. In *IEEE/ACM ICPC*. 249–259.

[31] LLVM. 2021. libFuzzer. https://llvm.org/docs/LibFuzzer.html#corpus.

[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *USENIX Security*. 1949–1966.

[33] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE TSE* (2019).

[34] K Paul Nesselroade Jr and Laurence G Grimm. 2018. *Statistical applications for the behavioral and social sciences.* John Wiley & Sons.

[35] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level directed fuzzing for use-after-free vulnerabilities. In *RAID*. 47–62.

[36] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Security*. 2289–2306.

[37] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *USENIX Security*. 729–743.

[38] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *IEEE/IFIP DSN*. 605–616.

[39] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *USENIX Security*. 861–875.

[40] sqlite. 2021. SQLite Source Repository. https://github.com/sqlite/sqlite/tree/master/test.

[41] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *ESORICS*. 581–601.

[42] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *ACM/IEEE ICSE*. 999–1010.

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *IEEE S&P*. 579–594.

[44] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS.

[45] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *ACM/IEEE ICSE*. 765–777.

[46] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *ACM SIGSAC CCS*.

[47] Dingning Yang, Yuqing Zhang, and Qixu Liu. 2012. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *IEEE TrustCom*. 1070–1076.

[48] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security*. 2307–2324.

[49] Michal Zalewski. 2017. afl-cmin. https://github.com/mirrorer/afl/blob/master/afl-cmin.

[50] Michal Zalewski. 2020. afl-tmin. https://github.com/google/AFL/blob/master/afl-tmin.c.

[51] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security*. 2255–2269.