Facilitating Parallel Fuzzing with Mutually-Exclusive Task Distribution

Yifan Wang[1], Yuchen Zhang[1], Chenbin Pang[2], Peng Li[3], Nikolaos Triandopoulos[1], and Jun Xu[1(✉)]

[1] Stevens Institute of Technology, Hoboken, USA
jxu69@stevens.edu
[2] Nanjing University, Nanjing, China
[3] ByteDance, Beijing, China

Abstract. Fuzz testing, or fuzzing, has become one of the de facto standard techniques for bug finding in the software industry. In general, fuzzing provides various inputs to the target program with the goal of discovering un-handled exceptions and crashes. In business sectors where the time budget is limited, software vendors often launch many fuzzing instances in parallel as a common means of increasing code coverage. However, most of the popular fuzzing tools—in their parallel mode—naively run multiple instances concurrently, without elaborate distribution of workload. This can lead different instances to explore overlapped code regions, eventually reducing the benefits of concurrency. In this paper, we propose a general model to describe parallel fuzzing. This model distributes mutually-exclusive but similarly-weighted tasks to different instances, facilitating concurrency and also fairness across instances. Following this model, we develop a solution, called AFL-EDGE, to improve the parallel mode of AFL, considering *a round of mutations to a unique seed* as a task and adopting edge coverage to define the uniqueness of a seed. We have implemented AFL-EDGE on top of AFL and evaluated the implementation with AFL on 9 widely used benchmark programs. It shows that AFL-EDGE can benefit the edge coverage of AFL. In a 24-h test, the increase of edge coverage brought by AFL-EDGE to AFL ranges from 9.5% to 10.2%, depending on the number of instances. As a side benefit, we discovered 14 previously unknown bugs.

Keywords: Software testing · Parallel fuzzing · Performance

1 Introduction

Thanks to its direct and easy application to production-grade software without human aids, fuzzing is gaining tremendous popularity for security testing. In

Y. Wang and Y. Zhang—These authors contributed equally.
C. Pang—This work was done while Pang was a Visiting Scholar at Stevens Institute of Technology.

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 185–206, 2021.
https://doi.org/10.1007/978-3-030-90022-9_10

today's business sectors, software systems are having shorter testing cycles [34], and therefore, the efficiency of code coverage becomes a critically desired property of fuzzing.

To escalate code coverage efficiency, there are two orthogonal strategies, one improving algorithms of fuzzing tools and one launching many fuzzing instances in parallel. The research community has intensively investigated the first strategy. Efforts along this line have revolutionized fuzzing from being program-structure-agnostic and black-box [2,5,28] to be program-structure-aware and grey-box/white-box [16,37,40,49,52], which significantly improved fuzzing efficiency by overcoming common barriers of code coverage.

However, the second strategy has been less studied and insufficiently developed. Existing fuzzing tools (*e.g.*, [8,37]) primarily follow American Fuzzy Lop (AFL) [52] to implement their parallel mode. Technically speaking, they run multiple identical instances in parallel. Depending on the implementation, different instances may either share the same group of seed inputs (or *seeds*) [8,37] or use separate groups of seed inputs but make periodical exchanges [52]. This type of parallel fuzzing, due to lack of synchronizations, leads different instances to run overlapped tasks, impeding the effectiveness of concurrency.

In this paper, we focus on unveiling the limitations of the parallel mode in existing fuzzing tools and presenting new solutions to overcome those limitations. We start with an empirical study of the parallel mode in AFL. By tracing the exploration of all instances across the fuzzing process, we discover that different instances are indeed running overlapped tasks despite many tasks remain unaccomplished (Sect. 2.2). We further demonstrate that this type of task overlapping can lead to reduced efficiency of code coverage.

Motivated and inspired by our empirical study, we propose a general model to describe parallel fuzzing. At the high level, the model enforces three desired properties. First, it distributes mutually-exclusive tasks to different instances, preventing the occurrence of overlaps. Second, it ensures every single task to be covered by at least one instance. This avoids the loss of fuzzing tasks and the code covered by those tasks. Finally, it assigns to each instance tasks with a similar amount of workload. Otherwise, some instances will be overloaded while the other instances are under-loaded, which can eventually degrade concurrency.

Guided by the model above, we develop a solution, called AFL-EDGE, towards facilitating the parallel mode in AFL. Our solution defines *a task* is to *run a round of mutations to a unique seed* and considers the control-flow edges (or *edges*) covered by a seed to determine its uniqueness. During the course of fuzzing, AFL-EDGE periodically distributes seeds that carry non-overlapped and similarly-weighted tasks to different instances, meeting the properties of our model. AFL-EDGE also enforces that all the unique seeds will cover the same set of edges as the original seeds. We envision that, in this way, AFL-EDGE can properly preserve the fuzzing capacity of AFL.

We have implemented AFL-EDGE on top of AFL, and we have evaluated AFL-EDGE with AFL using 9 widely adopted benchmark programs. Our evaluation shows that AFL-EDGE can significantly reduce the overlaps and hence, benefit the code coverage. Depending on the number of instances we launch,

we can averagely reduce 57.1%–60.3% of the overlaps and bring a 9.5%–10.2% increase in code coverage with AFL. Our evaluation also demonstrates that, compared to the state-of-the-art solutions of improving parallel fuzzing [22,39], our solution not only brings higher improvement to efficiency of edge coverage but also better preserves the capacity of the fuzzing tools. As a side benefit, AFL-EDGE triggers over 6K unique crashes, corresponding to 14 new bugs.

Our main contributions are as follows.

- We present a general model to describe parallel fuzzing.
- We develop a solution to improve the parallel mode in AFL, following the guidance of our model.
- We have implemented our solution on top of AFL, which can seamlessly run with other fuzzing tools that also use AFL. Source code of our implementation will be made publicly available upon publication.
- We evaluated our solution with AFL on 9 widely used benchmark programs. It shows that our solution can effectively reduce the overlaps and increase the code coverage of AFL.

2 Background and Motivation

2.1 Grey-Box Fuzzing and Parallel Mode

In this research, we target grey-box fuzzing [25], the most popular category of fuzzing. Grey-box fuzzing generally follows the *feedback-scheduling-mutation* model presented in Fig. 1. This FSM model represents an iterative process, starting with a queue of seed inputs, or *seeds*, that are typically generated from certain known test cases. In a round of fuzzing, the *scheduling* picks a seed and feeds it to the *mutation* process for deriving new inputs to test the target program, expecting to trigger un-handled crashes or exceptions. Both the scheduling and mutation processes are based on *feedback* (e.g., crashes and code coverage) obtained from the program executions on the previously generated inputs. The fuzzer also collects feedback to decide whether an input under test should be added to the seed queue.

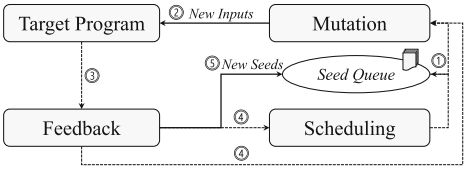
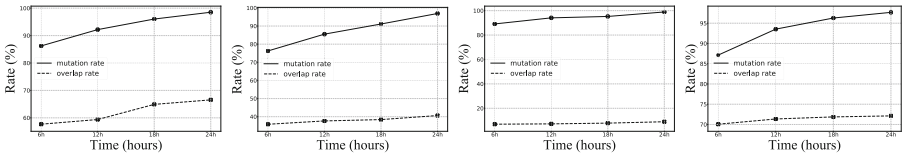


Fig. 1. A general model of grey-box fuzzing.

To improve the efficiency of code coverage, many grey-box fuzzing tools [8, 37,52] provide a parallel mode to run multiple instances concurrently. Their parallel mode mostly follows AFL. They start identically-configured instances and run them in parallel. Depending on the implementation, different instances may either share the same seed queue [37] or carry separate seed queues but periodically exchange seeds [52]. In the latter case, each instance borrows from other instances all the seeds that bring new code coverage. While intuition suggests that more fine-grained synchronizations can benefit the effectiveness of the above parallel fuzzing, none of the existing tools carry such synchronizations.



(a) OBJDUMP (b) READELF (c) LIBXML (d) NM-NEW

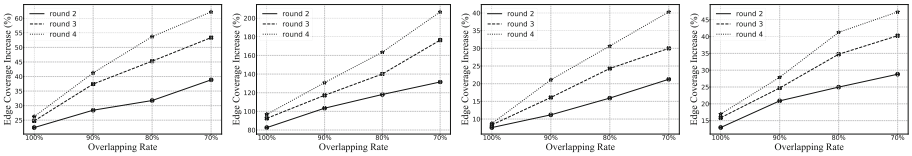
Fig. 2. Results of measuring overlapped mutations: "mutation rate" - the portion of seeds mutated by at least one instance; "overlap rate" - the portion of seeds mutated by more than one instance.

2.2 Motivating Study

Sharing seeds across instances is an intentional design of AFL [51]. The goal is that "hard-to-hit but interesting test cases" can be used by all instances to guide their work. However, intuition suggests that such a design can lead different instances to mutate the same seeds, which may eventually reduce the effectiveness of concurrency. To validate this intuition and thus, motivate our research, we perform an empirical study based on AFL.

In our study, we run AFL on four popular benchmark programs (OBJDUMP, READELF, LIBXML, NM-NEW) with 2 parallel instances for 24 h. We trace the mutation process to understand which seeds are mutated by which instances. We repeat the tests five times and report the average results in Fig. 2. As shown by the results, different instances are indeed mutating overlapped seeds despite many seeds are never mutated, in particular when the fuzzing time is limited. Consider the results after 6 h as an example. On average, nearly 20% of the seeds are never mutated. However, over 42% of the seeds receive multiple rounds of mutations. When we increase the fuzzing time, we observe even higher rates of overlaps but still a group of non-mutated seeds.

Although we observe overlaps among mutated seeds and the overlaps indeed delay the mutations to all seeds, they may not necessarily impede the efficiency of code coverage. This is because AFL's mutation involves random operations. In this regard, running multiple rounds of mutations to the same seed—especially when the seed has higher potential—may produce code coverage comparable to applying the mutations to different seeds. To verify this possibility, we run another experiment. Specifically, we randomly collect 1,000 seeds produced in the above test of each program and equally split the seeds into group `A` and group `B`. First, we run AFL to mutate group `A` for multiple rounds and we calculate the increase of code coverage after each round of mutations, where we use control flow edges as the metric of code coverage and we consider the edges covered by the original 1,000 seeds as the baseline. Then, we repeat the experiment but replace the same $X\%$ of group `A` with un-mutated seeds from group `B` in each round. For example, when we set X to 10, we run the original group `A` in the first round but in every following round, we replace a fixed subset of 50 seeds with other non-muted ones from group `B`. Such an experiment enables us to simulate fuzzing scenarios where a different extent of overlapped mutations happen.



(a) OBJDUMP (b) READELF (c) LIBXML (d) TIFF2PS

Fig. 3. Impacts of overlapped mutations to edge coverage. The baseline of "Edge Coverage Increase" is the edges covered by the original 1,000 seeds; "Overlapping Rate" indicates the portion of overlapped seeds between two consecutive rounds of mutations. Results from the first round are omitted since the first round is identical under different settings, i.e., mutating the initial 500 seeds. *Please note that the x-axis decreases from left to right.*

Figure 3 presents the results of the above experiment under different settings. With all the four programs, we observe a trend that fewer overlaps among the mutated seeds lead to a higher increase of code coverage. This empirically demonstrates that running AFL's mutations on different seeds can cover more edges than running the mutations on the same seeds. We believe such results align with AFL's design: AFL distributes mutation energies in a round according to the potential of a seed (based on metrics such as number of edges covered by the seed) and assigns more mutation cycles to seeds with higher potential, which eventually helps allocate sufficient mutations in a single round to exhaust the edges that can be derived from a seed.

To sum up, our empirical study shows that the parallel mode of AFL (and likely, many other fuzzing tools) can indeed bring overlaps, which further impedes the efficiency of code coverage. It is, therefore, necessary to investigate and develop better solutions of parallel fuzzing.

3 A General Model of Parallel Fuzzing

In this section, we propose a model to describe parallel fuzzing. The model is inclusive of the parallel mode in existing fuzzing tools and we envision it is general enough to apply to other solutions of parallel fuzzing.

Formally, a parallel fuzzing system consists of n instances $\{F_1, F_2, ..., F_n\}$. These instances together work on a set of m tasks $\{T_1, T_2, ..., T_m\}$, with the i^{th} instance distributed to focus on a subset of tasks $\{T_{i1}, T_{i2}, ..., T_{im_i}\}$ $(m_i \leq m)$. Depending on the definition of tasks, $\{T_1, T_2, ..., T_m\}$ can require different amounts of workload to be completed, notated as $\{W_1, W_2, ..., W_m\}$. To improve the efficiency of parallel fuzzing, the system would desire to meet the following three properties.

- $(\mathbb{P}_1)$ Different instances should work on disjoint subsets of tasks. This is to avoid overlaps and increase the extent of concurrency. Formally, given any two instances F_i and F_j $(i \neq j)$, the fuzzing system needs to ensure:

$$\{T_{i1}, T_{i2}, ..., T_{im_i}\} \cap \{T_{j1}, T_{j2}, ..., T_{jm_j}\} = \emptyset \tag{1}$$

- ($\mathbb{P}_2$) All the instances together should cover all the tasks. Formally, that means:
$$\bigcup_{i=1}^{n}\{T_{i1}, T_{i2}, ..., T_{im_i}\} = \{T_1, T_2, ..., T_m\} \tag{2}$$
Otherwise, the pursuit of parallel fuzzing can cause the loss of certain tasks and, essentially, miss the code that can be covered by those tasks.
- ($\mathbb{P}_3$) Different tasks should be assigned with a similar workload. Formally, the fuzzing system should maintain the following relation between any two instances F_i and F_j ($i \neq j$):
$$\sum_{n=1}^{m_i} W_{in} \approx \sum_{n=1}^{m_j} W_{jn} \tag{3}$$
Otherwise, certain instances can receive under-loaded tasks and end with plenty of idle cycles, which in principle also harms concurrency.

While the above model is general, it overlooks the fact that different tasks can bring different benefits, in particular code coverage [9]. To this end, Eq. 1 should be amended to allow overlaps of tasks that carry higher returns such that these tasks have a higher chance to be picked and completed. To incorporate this consideration, we find that it is not mandatory to modify our model. Instead, we can replicate high-return tasks and consider their replicas as unique ones. This will achieve similar effects as allowing overlaps of those tasks.

4 Applications of Our Model

4.1 Existing Solutions

In the literature, two solutions of improving parallel fuzzing, P-FUZZ [39] and PAFL [22], fit into our model. Both P-FUZZ and PAFL consider *a fuzzing task is to run a round of mutations to a unique seed* and they distribute a similar amount of unique seeds to each instance. However, the two solutions take two opposite principles to define the uniqueness of a seed. P-FUZZ aims for conservativeness. It follows AFL and considers a seed that brings new code coverage at its birth to be unique. Such a principle preserves all seeds produced by AFL but can leave behind many overlaps. Consider Fig. 4, where `Seed-1` is born

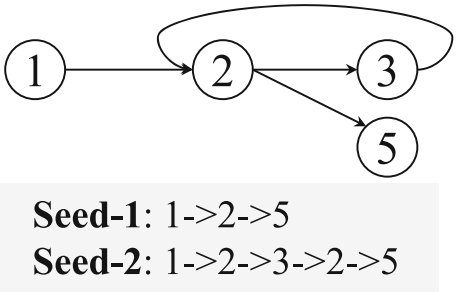


Fig. 4. An example of two seeds that cover overlapped sets of edges. The *upper* part presents the CFG; the *lower* part shows the two seeds.

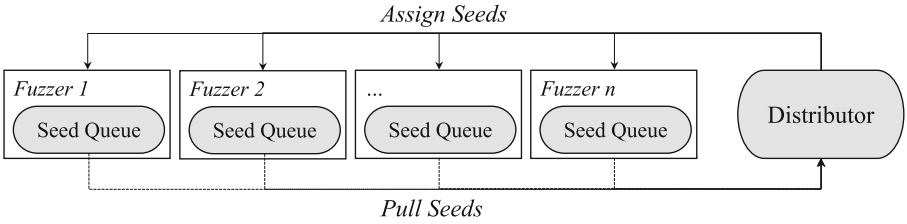


Fig. 5. Workflow of our solution to optimize parallel fuzzing.

before `Seed-2`, as an example. P-FUZZ will consider both seeds unique and mutate both seeds. However, `Seed-2` covers all the edges of `Seed-1` and thus, they actually overlap based on AFL's definition. Moreover, mutating `Seed-2` can likely produce the same code coverage as mutating both seeds, further illustrating the overlap.

In contrast, PAFL aims for effectiveness. It considers a seed unique only when the seed covers certain less-frequently visited edges. In this way, PAFL can massively reduce overlaps. However, it does not guarantee that the unique seeds can cover all the code covered by the original seeds. As such, it can skip certain code regions and lose the opportunities to cover code that can be derived from those code regions.

4.2 Our Solution

In this paper, we propose a new solution following our model. Similar to P-FUZZ and PAFL, we consider a fuzzing task is to run a round of mutations to a unique seed. However, as we will explain shortly, we adopt a strategy that achieves an effectiveness-and-conservativeness balance to define the uniqueness of a seed. Our solution follows the workflow in Fig. 5 to periodically distribute the tasks. In each round of task distribution, we pull seeds from all instances, partition them into un-overlapped while similarly-weighted sub-sets, and finally assign them back to each instance. In the rest of this section, we describe the design details and explain how they meet $\mathbb{P}_1$ - $\mathbb{P}_3$.

Defining Fuzzing Tasks. In our solution, we consider the entire set of tasks are to mutate seeds that cover all the (control flow) edges reached by the original seeds. In this way, we approximate the fuzzing goals of AFL and largely preserve the fuzzing space of AFL (i.e., produce similar effects as mutating every seed). To determine the uniqueness of a seed, we consider the edges and their hit counts[1] covered by the seed as criteria. But different from P-FUZZ, we consider *a seed is unique only if the seed covers one or more edges that other seeds do not cover.* This principle avoids the overlaps that P-FUZZ may incur. Referring back to the example in Fig. 4, when `Seed-1` and `Seed-2` both exist at the moment of

[1] A hit count in each of the following ranges is mapped to a unique value: `[1]`, `[2]`, `[3]`, `[4, 7]`, `[8, 15]`, `[16, 31]`, `[32, 127]`, `[128,` ∞`)`.

Algorithm 1: TASK DISTRIBUTION

Input : Seed sets from all instances $\mathcal{D} = \{\vec{S}_1, \vec{S}_2, ..., \vec{S}_n\}$
Output : Seeds distributed to different instances $\mathcal{D}' = \{\vec{S}'_1, \vec{S}'_2, ..., \vec{S}'_n\}$

Initialize $\mathcal{D}'$: $\vec{S}'_1 = \emptyset$, $\vec{S}'_2 = \emptyset$,... $\vec{S}'_n = \emptyset$
for *each* $\vec{S}_i \in \mathcal{D}$ **do**
  Obtain edges covered by seeds in $\vec{S}_i$, notated as $\vec{E}_i$;
  /* hit counts of edges are considered */
end
$\vec{E} = \bigcap_{i=1}^{n} \vec{E}_i$;
Organize $\vec{E}$ into a control flow graph CFG;
/* different hit counts of the same edge are represented as different edges */
Copy CFG as CFG' and topologically sort CFG';
for *the deepest leaf node* L_i *in* CFG' **do**
  $k = random(1, n)$;
  Pick a seed s from $\vec{S}_k$ which covers L_i, maximizes $|edge(s) \bigcap edge(CFG')|$, and has the minimal age;
  Add s to $\vec{S}'_k$;
  Remove $edge(s)$ from CFG';
end
for *each* $\vec{S}_i \in \mathcal{D}$ **do**
  for *each* $s \in \vec{S}_i$ **do**
    if $edge(s) - edge(CFG) \neq \emptyset$ *and* $edge(s) - edge(\vec{S}'_i) \neq \emptyset$ **then**
      Add s to $\vec{S}'_i$;
    end
  end
end
return $\mathcal{D}'$;

task distribution, we will consider `Seed-1` non-unique since `Seed-2` encapsulates all the edges of `Seed-1`. In the following, we describe how to pick unique seeds while running task distribution.

Distributing Fuzzing Tasks. In each round of task distribution, we pull all the seeds from each instance and re-run them with dynamic tracing. We gather the edges covered by each instance, notated as $\vec{E}_i$ for the i^{th} instance. We then compute the intersections among the edges from all instances (i.e., $\bigcap_{i=1}^{n} \vec{E}_i$), and we notate the intersections as $\vec{E}$. By intuition, we can then randomly, evenly partition $\vec{E}$ into multiple sub-sets, assign each sub-set to a unique instance, and then pick seeds that visit those assigned edges for the instance to mutate. Such an idea, however, has a major problem. When we pick a seed to cover a particular edge, we will concurrently cover many other edges, which may essentially bring overlaps back. Consider the Fig. 6 as an example. By random distribution, we may sequentially pick `Seed-2`, `Seed-3`, and `Seed-4`, and distribute them to different instances.

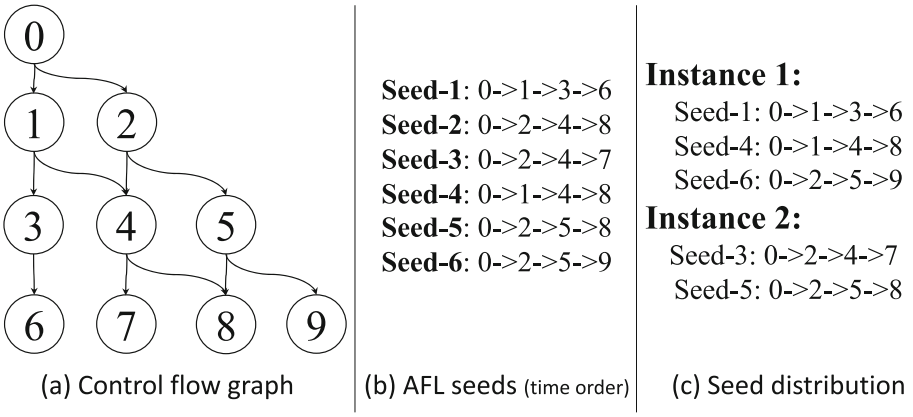


Fig. 6. An example of edge-coverage-based task distribution between 2 instances. The *left* part shows the CFG aggregated by the overlapped edges. The *middle* part show the seeds produced by AFL, sorted in time order. The *right* part presents the task distribution results.

According to our definition, this would create an edge-overlap between `Seed-2` and `Seed-3` + `Seed-4`, not satisfying our definition of uniqueness.

In this work, we design a greedy algorithm, shown in Algorithm 1, to provide *edge-coverage-based task distribution*. We aggregate the edges in $\vec{E}$ into a topologically sorted control flow graph, notated as CFG (line 1-6). We then recursively process the leaf edges on CFG (i.e., edges that end with leaf nodes on CFG). For the leaf edge with the largest depth, we randomly pick a fuzzing instance and elaborately pick a seed s from that instance to cover the leaf edge (line 9-10). To be specific, we select the seed that covers the maximal number of edges remaining on the CFG. If multiple seeds satisfy this condition, we pick the youngest one. We distribute s to the fuzzing instance where s comes from (line 11) and remove all edges covered by s from CFG (line 12). We repeat this process until all edges on the CFG are removed. After that, we preserve the seeds that visit edges in $\neg\vec{E}$ (line 14-20). To better illustrate our distribution algorithm, we present an example in Fig. 6, showing both the fuzzing progress and the distribution results. It is worth noting that when we pick a seed for leaf node `4`→`8`, we favor `Seed-4` over `Seed-2` because `Seed-4` is more recently derived. This prevents the pick of `Seed-2` and avoids the overlap we mentioned before.

The above algorithm involves multiple heuristics, which strive for fewer overlaps and better efficiency. First, we prefer seeds that cover more non-distributed edges (line 10). The motivation is to quickly consume the distribution space and thus, minimize the number of required seeds and reduce the potential of overlaps. Second, we favor newer seeds. The rationale is that seeds newly generated have a higher chance to cover new edges than the older seeds. Thus, they have a lower chance of bringing in overlaps. Third, we prioritize edges that have larger

depths on the control flow graph. This is to reduce the search space when picking seeds, exploiting the observation that deeper edges are typically reached by fewer seeds.

Despite our greedy algorithm may not perfectly meet $\mathbb{P}_1$ - $\mathbb{P}_3$, it represents the best effort. First, we distribute disjoint sub-sets of overlapped edges to different instances. We further incorporate a set of heuristics to avoid overlaps when we pick seeds. As we will demonstrate in Sect. 6, this combined effort can indeed effectively reduce overlaps (way more than P-FUZZ). Second, every edge (in $\vec{E}$ or $\neg\vec{E}$) is distributed to at least one instance. This preserves all the fuzzing tasks according to our definition, satisfying $\mathbb{P}_2$. Finally, we evenly distribute the non-overlapped edges in a random manner. This aids each instance to receive approximately equivalent workloads and, therefore, facilities the fulfillment of $\mathbb{P}_3$.

Scheduling Task Distribution. Our design needs to periodically re-run the task distribution. However, a low frequency of re-distribution may not timely avoid the accumulated overlaps while a high frequency can lead to a waste of computation cycles since fuzzing may not have produced many overlaps. In our design, we adjust the scheduling of task distribution based on the increase of edge coverage. We start the first round of distribution after the first hour, and we re-run it once the new edge coverage exceeds 10%.

5 Implementation

We have implemented our solution, called AFL-EDGE, on top of AFL (2.52b) and LLVM with around 100 lines of C code, 400 lines of C++ code, 300 lines of shell scrips, and 200 lines of Python code. All code will be released upon publication.

5.1 Collecting Edge Coverage

The task distribution of AFL-EDGE needs code coverage information of existing seeds. To support the need, we implement an LLVM pass to instrument the target program. Following a seed, the instrumented code will sequentially record each edge and output the final list at the end. To avoid collisions, we assign each basic block a unique 64-bit ID and concatenate the IDs of two connected basic blocks to represent the edge between them.

5.2 Distributing Fuzzing Tasks

AFL-EDGE requires to distribute seeds across fuzzing instances. To avoid intruding on the normal fuzzing process, we implement the task distributor as a standalone component. It follows the algorithm in Sect. 4.2 to determine the seeds that are assigned to each instance and saves the seeds in a file. Following the metadata organization of AFL, the seed file is added to the corresponding instance's working directory.

Table 1. Benchmark programs and evaluation settings. In the column of `Seeds`, `AFL` means we reuse the test-cases from AFL and `built-in` means that we reuse the test cases from the program.

Programs				Settings	
`Name`	`Version`	`Driver`	`Source`	`Seeds`	`Options`
LIBPCAP	1.10.0	TCPDUMP	[41]	AFL	-r @@
LIBTIFF	4.0.10	TIFF2PS	[19]	AFL	@@
LIBTIFF	4.0.10	TIFF2PDF	[19]	AFL	@@
BINUTILS	2.32	OBJDUMP	[15]	AFL	-d @@
BINUTILS	2.32	READELF	[15]	AFL	-a @@
BINUTILS	2.32	NM-NEW	[15]	AFL	-a @@
LIBXML2	2.9.7	XMLLINT	[26]	AFL	@@
NASM	2.14.2	NASM	[3]	built-in	-e @@
FFMPEG	4.1.1	FFMPEG	[6]	built-in	-i @@

5.3 Confining Fuzzing Tasks

Our design requires an instance to only mutate the sub-group of assigned seeds. Technically, we customize AFL to read the list of seeds assigned by the distributor and maintain them in a allow-list. When AFL schedules seeds for mutations, we only pick candidates on the allow-list. Such an implementation avoids introducing extra inconsistency to the fuzzing process. Considering that our distributor iteratively updates the seed list, the customized AFL periodically checks the seed file and updates the allow-list accordingly.

6 Evaluation

In this section, we evaluate AFL-EDGE, centering around three questions.

- ($\mathbb{Q}_1$) *Can* AFL-EDGE *reduce the overlaps among fuzzing instances?*
- ($\mathbb{Q}_2$) *Can* AFL-EDGE *improve the efficiency of code coverage?*
- ($\mathbb{Q}_3$) *Can* AFL-EDGE *preserve the fuzzing capacity of* AFL*?*

6.1 Experimental Setup

Benchmarks. To answer the above questions, we prepare a group of 9 real-world benchmark programs. Details about the programs are presented in Table 1. All these programs have been intensively tested in both industry [43] and academia [32,40,49]. In addition, they carry diversities in both functionality and complexity.

Baselines. We run AFL as the baseline of our evaluation. To compare AFL-EDGE with the existing solutions, we also run P-FUZZ [39] and PAFL [22]

on top of AFL. Because the implementations of P-FUZZ and PAFL are not publicly available, we re-implemented the two solutions following the algorithms presented in their publications [22,39].

Configurations. Specific configurations of the fuzzing process (e.g., seeds and program options) are listed in Table 1. To understand the impacts of the number of instances, we run each fuzzing setting respectively with 2, 4, and 8 AFL secondary instances. We do not run a primary instance because it involves deterministic mutations which bring disadvantages to vanilla AFL. For consistency, we conduct all the experiments on Amazon EC2 instances (Intel Xeon E5 Broadwell 96 cores, 186GB RAM, and Ubuntu 18.04 LTS), and we sequentially run all the tests to avoid interference. Each test is run for 24 h. To minimize the effect of randomness in fuzzing, we repeat each test 5 times and report the average results.

6.2 Analysis of Results

In Table 2, we present the results with AFL at the end of 24 h. We elaborate on the results as follows, seeking answers to $\mathbb{Q}_1$ - $\mathbb{Q}_3$.

Effectiveness of Overlap Reduction. The direct goal of AFL-EDGE is to reduce the overlaps among instances. To measure this goal, we consider the number of seeds that are disabled from each instance as the metric. As shown in Table 2 (the column for *overlap reduction rate*), AFL-EDGE can effectively reduce the potential overlaps in the parallel mode of AFL. To be specific, AFL-EDGE can prevent 60.0%, 60.3%, and 57.1% of the seeds from being repeatedly mutated when we respectively run 2, 4, and 8 parallel instances.

In comparison to existing solutions, AFL-EDGE reduces more overlaps than P-FUZZ but fewer than PAFL. Such results well comply with the designs of the three tools. P-FUZZ preserves all the seeds produced by AFL while PAFL aggressively skip seeds. In contrast, AFL-EDGE keeps seeds necessary to cover all the edges, pursuing a trade-off between conservativeness and effectiveness. As we will show later, while AFL-EDGE's strategy reduces fewer seeds in comparison to PAFL, it does not necessarily hurt code coverage and it can better preserve the fuzzing capacity (or more precisely, AFL-EDGE can cover more code that AFL covers).

Improvements to Code Coverage Efficiency. To understand whether the overlap reduction by AFL-EDGE can indeed benefit code coverage, we measure the number of edges covered in the tests. In Table 2 (the column of *edge coverage increase*), we present the increase of edge coverage brought by AFL-EDGE to AFL at the end of a 24-h test.

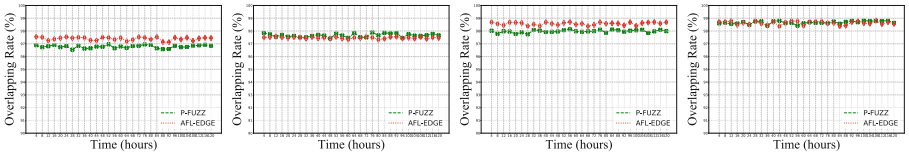
In summary, AFL-EDGE can consistently improve the efficiency of edge coverage of AFL, regardless of the benchmark and the number of instances. Specifically, AFL-EDGE increases the edge coverage by 10.0%, 10.2%, and 9.5%, respectively with 2, 4, and 8 instances. Another key observation is that the benefits brought by AFL-EDGE often decrease with the number of instances. We

Table 2. Statistical results of our evaluation with AFL in 24 h. In the table, "overlap reduction (%)" means the average percentage of seeds that the corresponding solution cuts from each instance; "edge-cov increase (%)" stands for the increase of code coverage that the corresponding solution brings to AFL; p-value demonstrates the statistical significance of the increase of code coverage (the smaller, the better); and "edge-cov overlap rate (%)" shows how much of the code covered by AFL is also covered by the corresponding solution.

Prog.	Tool	Statistical evaluation results with 2, 4, and 8 instances (AFL).											
		overlap reduction (%)			edge-cov increase (%)			p-value			edge-cov overlap (%)		
OBJDUMP	PAFL	60.6	80.8	89.5	1.9	0.0	0.3	0.32	0.72	0.10	95.5	94.9	95.8
	P-FUZZ	46.7	53.5	62.5	5.1	2.3	3.0	0.07	0.03	0.00	96.8	96.3	97.9
	AFL-EDGE	63.7	63.2	62.4	7.6	7.3	3.1	0.00	0.04	0.03	97.5	98.3	98.5
READELF	PAFL	81.2	86.8	88.4	5.6	-3.2	-3.6	0.03	0.19	0.12	92.5	92.1	93.0
	P-FUZZ	46.8	43.8	45.4	7.2	3.8	6.0	0.04	0.01	0.00	99.0	98.5	98.0
	AFL-EDGE	45.9	43.8	42.5	12.2	7.0	8.4	0.05	0.02	0.00	97.5	97.2	97.7
TIFF2PDF	PAFL	64.1	79.7	81.2	3.4	0.4	5.0	0.00	0.05	0.29	97.0	91.3	93.8
	P-FUZZ	42.2	60.1	64.1	5.5	6.3	8.5	0.01	0.04	0.04	98.5	97.0	97.9
	AFL-EDGE	62.4	59.5	58.5	9.0	5.6	11.2	0.00	0.02	0.00	98.5	95.9	99.1
NM-NEW	PAFL	46.2	52.6	55.9	4.0	4.8	0.2	0.10	0.07	0.19	96.6	96.2	97.5
	P-FUZZ	43.6	55.6	60.2	4.5	7.1	3.8	0.00	0.03	0.00	97.6	98.1	98.1
	AFL-EDGE	59.9	58.5	60.1	6.8	6.6	3.6	0.03	0.08	0.00	96.3	96.8	98.5
NASM	PAFL	64.2	82.7	54.7	5.5	14.6	2.4	0.05	0.01	0.06	99.8	99.3	99.8
	P-FUZZ	10.0	11.0	8.2	8.9	9.2	8.1	0.00	0.00	0.00	99.0	98.3	94.3
	AFL-EDGE	67.9	68.3	66.8	13.9	22.6	7.6	0.00	0.00	0.00	99.2	99.3	99.2
TIFF2PS	PAFL	57.1	81.9	89.2	7.8	15.0	8.7	0.03	0.02	0.02	82.3	89.9	96.6
	P-FUZZ	44.0	60.7	66.4	7.5	12.3	9.5	0.00	0.00	0.00	97.1	97.5	98.0
	AFL-EDGE	53.4	58.5	52.8	10.4	13.5	7.2	0.00	0.00	0.00	96.9	96.5	97.2
TCPDUMP	PAFL	66.4	80.3	84.4	8.2	10.6	7.7	0.19	0.03	0.06	84.7	86.6	88.5
	P-FUZZ	45.6	63.6	80.7	2.8	7.2	6.1	0.05	0.03	0.04	85.7	88.3	91.5
	AFL-EDGE	48.3	48.6	42.9	4.4	9.9	11.9	0.05	0.03	0.00	87.6	89.5	92.8
LIBXML2	PAFL	61.8	91.5	82.2	7.3	43.7	7.0	0.00	0.01	0.06	87.2	81.7	88.3
	P-FUZZ	41.0	48.5	51.8	5.1	11.7	21.3	0.00	0.01	0.00	97.9	89.3	97.6
	AFL-EDGE	68.9	69.2	65.6	7.5	7.6	29.1	0.00	0.04	0.00	98.6	93.3	98.2
FFMPEG	PAFL	94.5	90.7	91.4	17.9	23.4	4.1	0.04	0.01	0.38	86.8	85.2	87.0
	P-FUZZ	41.6	62.5	63.4	22.6	20.1	6.1	0.03	0.05	0.00	91.2	90.2	98.3
	AFL-EDGE	69.7	73.5	62.3	18.3	11.6	3.3	0.01	0.04	0.00	89.9	90.1	97.3
Ave.	PAFL	**66.2**	**80.8**	**79.7**	**6.9**	**12.1**	**3.5**	–	–	–	**91.4**	**90.8**	**93.4**
	P-FUZZ	**40.2**	**45.1**	**49.5**	**7.7**	**8.9**	**8.0**	–	–	–	**95.9**	**94.8**	**96.9**
	AFL-EDGE	**60.0**	**60.3**	**57.1**	**10.0**	**10.2**	**9.5**	–	–	–	**95.8**	**95.2**	**97.6**

believe the reason is that the fuzzers can get closer to saturation when more parallel instances are running. Therefore, the gap between AFL and AFL-EDGE shrinks at the end.

To further verify that the improvements by AFL-EDGE are statistically significant, we perform Mann Whitney U-test [27] on the five rounds of runs [18].



(a) OBJDUMP (b) READELF (c) LIBXML (d) TIFF2PDF

Fig. 7. Overlap of code coverage between AFL-EDGE/P-FUZZ and AFL in the 120-h tests.

The p-values of the hypothesis test are presented in Table 2 (the column of *p-value*). In nearly all the cases, the p-values are smaller than 0.05, supporting that the improvements brought by AFL-EDGE are significant from a statistical perspective.

Finally, AFL-EDGE presents better overall performance than both P-FUZZ and PAFL. When applied to AFL, AFL-EDGE increases the edge coverage by 9.5% - 10.2%, outperforming P-FUZZ and PAFL in most of the cases. On the one hand, AFL-EDGE reduces more overlaps than P-FUZZ and thus, produces higher code coverage efficiency. On the other hand, PAFL in principle reduces more overlaps than AFL-EDGE, which indeed leads to higher edge coverage than AFL-EDGE in several cases (e.g., running AFL on TIFF2PS with 4 or 8 instances). However, in many other cases, PAFL can accidentally block valuable seeds and become unable to cover the related edges, eventually resulting in a lower edge coverage. This can be further supported by that PAFL even produces lower edge coverage than AFL in certain cases (e.g., running AFL on READELF with 4 or 8 instances).

Effectiveness of Preserving Fuzzing Capacity. As discussed in Sect. 4, AFL-EDGE can skip certain seeds produced by AFL. This may alter the fuzzing behaviors and, more concerningly, hurt the fuzzing capacity (i.e., missing edges that can be covered by vanilla AFL). To understand the impacts of AFL-EDGE to the fuzzing capacity, we perform another analysis where we examine whether AFL-EDGE and AFL are exploring different edges. Technically, we measure how many of the edges covered by AFL are also covered by AFL-EDGE. We show the results in Table 2 (the column of *edge overlap rate*). In summary, AFL-EDGE can prevalently cover more than 95% of the edges that are covered by AFL. Considering the existence of randomness, we believe such results strongly support that AFL-EDGE largely preserves the behaviors of the vanilla tools and does not significantly affect the fuzzing capacity.

In comparison to existing solutions (see Table 2), AFL-EDGE can preserve as much edge coverage as P-FUZZ. This further proves that AFL-EDGE well maintains the fuzzing space since P-FUZZ does not skip seeds and thus, its results represent the best efforts. Further, AFL-EDGE outperforms PAFL in covering the edges reached by AFL (96.2% v.s. 91.8%). This is because AFL-EDGE keeps seeds to cover all the original code to avoid losing fuzzing capacity while PAFL more aggressively skips seeds.

Table 3. Impacts of frequency of our task distribution.

Prog.	Setting	Number of edges covered in 24h			
		once/1h	*once/2h*	*once/4h*	*dynamic*
OBJDUMP	AFL-EDGE	33422	33828	33370	**34402**
READELF	AFL-EDGE	50932	53036	51927	**53839**

To validate the above observations in longer-term fuzzing, we extend the tests of AFL-EDGE to 120 h. We also run this test with P-FUZZ as a comparison. As shown in Fig. 7, AFL-EDGE consistently preserves the edges covered by AFL across the 120 h, producing results comparable to P-FUZZ. We note that in certain cases, AFL-EDGE even slightly outperforms P-FUZZ. This is mostly because AFL-EDGE has a higher efficiency of edge coverage than P-FUZZ and therefore, reaches more edges that AFL covers.

Impacts of Frequency of Task Distribution. Recall that AFL-EDGE needs to periodically distribute tasks (Sect. 4). Our hypothesis is that the frequency of distribution can affect the effectiveness of our solution and we dynamically adjust this frequency based on the growth of edges. To validate our hypothesis and demonstrate the utility of our dynamic approach, we perform another experiment where we run one round of distribution per 1 h, 2 h, and 4 h. In Table 3, we present the results. It shows that the frequency of distribution truly makes a difference and our dynamic adjustment indeed outperforms solutions with a fixed frequency.

Table 4. Comparison of seed distillation algorithms. The numbers show the amount of seeds picked by different algorithms.

Prog.	Number of seeds picked after distillation			
	`Unweighted`	`Weight-Time`	`Weight-Size (cmin)`	`Ours`
OBJDUMP	601	803	599	596
READELF	939	980	938	683
TCPDUMP	765	849	756	592
XML	689	862	686	434
NASM	490	724	492	452
NM	541	701	788	778
TIFF2PDF	785	920	778	778
TIFF2PS	822	916	817	802
FFMPEG	593	742	589	581

Effectiveness of Seed Distribution. In our algorithm of task distribution algorithm (Algorithm 1), the core idea is to pick a subset of seeds that cover the

original edges, commonly known as *seed distillation*. Past efforts have developed several other seed distillation algorithms, including AFL-CMIN [50] (notated as *Size-Weighted*) and its variants (including [1], notated as *Unweighted*; [33,45], notated as *Time-Weighted*). Details of the algorithms are as follows.

- **Unweighted Algorithm.** This algorithm always picks a seed whose edges overlap with the non-covered edges the most. It repeats until all edges are covered.
- **Time-Weighted.** This algorithm iterates each non-covered edge and picks the seed with the shortest execution time to cover the edge, repeating this process until all edges are covered.

Table 5. Unique crashes/bugs discovered in our tests.

Prog.	AFL		PAFL		P-FUZZ		AFL-EDGE		Bug types
	Crash	Bug	Crash	Bug	Crash	Bug	Crash	Bug	---
FFMPEG	12	1	72	1	54	1	672	1	Heap overflow
TIFF2PDF	0	0	10	1	2	1	99	1	Failed allocation
TIFF2PS	0	0	0	0	126	1	260	3	Heap overflow
NASM	631	2	1872	6	1,430	6	5,765	9	Memory leaks stack overflow
Total	**643**	**3**	**1954**	**8**	**1612**	**9**	**6792**	**14**	—

- **Size-Weighted (AFL-CMIN).** This algorithm iterates each non-covered edge and picks the seed with the smallest size to cover the edge, repeating this process until all edges are covered.

We conduct an experiment to compare our algorithm with the existing algorithms: we run these algorithms on 1,000 random seeds from each of our benchmark programs and count the number of picked seeds. As shown in Table 4, our algorithm reduces more seeds than all the existing algorithms in every benchmark program, demonstrating better effectiveness. Note that we skipped some other algorithms (e.g., [17]) as they cannot ensure all original edges (or hit counts of edges) are preserved.

6.3 Evaluation of Bug Finding

In the course of evaluation, the fuzzing tools also trigger many crashes. We triage these crashes with AddressSanitizer [36] and then perform a manual analysis to understand the root causes. As shown in Table 5, AFL-EDGE triggers 6,792 unique crashes and 14 previously unknown bugs, outperforming both P-FUZZ and PAFL. Moreover, all the bugs detected by AFL and P-FUZZ are also detected by AFL-EDGE.

We also extended the evaluation of bug finding with the LAVA vulnerability benchmark [12]. However, we omitted the reporting of the results. Basically,

our tests with AFL only trigger 1 LAVA bug, regardless of the parallel fuzzing solutions. The major reason is that all LAVA bugs require a four-byte unit in the input to match a random integer value, which is hard to be satisfied by AFL's mutations.

7 Related Works

7.1 Improvements to Algorithms of Grey-Box Fuzzing

Past research has brought three categories of algorithmic improvements to grey-box fuzzing. The first category explores new kinds of *feedback* to facilitate seed scheduling and mutation. AFL [52] considers code branches covered in a round of execution as feedback, which is further refined by Steelix [21], CollAFL [13], and PTrix [10] with more fine-grained, control-flow related information. TaintScope [44], Vuzzer [32], GREYONE [14], REDQUEEN [4], and Angora [8] use taint analysis to identify data flows that can affect code coverage.

The second category of research investigates how to use the above types of feedback to improve code coverage. FairFuzz [20], GREYONE [14], and ProFuzzer [48] rely on the feedback to mutate the existing inputs and derive new ones that have a higher probability of reaching new code. AFLFast [7], DigFuzz [53], and MOPT [23] consider the feedback as guidance to schedule inputs for mutation and prioritizes those with higher potentials of leading to new code.

The last category aims at improving *mutations* to remove common barriers that prevent fuzzers from reaching more code. Majundar et al. [24] introduce the idea of hybrid fuzzing, which runs concolic execution to solve complex conditions that are difficult for pure fuzzing to satisfy. The idea was followed and improved by many other works [30,40,49,53]. TFuzz transforms target programs to bypass complex conditions and forces the execution to reach new code. It then uses a validator to reproduce the inputs that meet the conditions in the original program. Angora [8] assumes a black-box function at each condition and uses gradient descent to find satisfying inputs, which is later improved by NEUZZ [38].

Differing from the above works, our research aims to improve the efficiency of the parallel mode of fuzzing, an orthogonal strategy to facilitate the efficiency of code coverage.

7.2 Improvements to Execution Speed of Fuzzing

Beyond algorithmic improvements, other research aims to improve the efficiency of fuzzing by accelerating the fuzzing execution. PTrix [10], Honggfuzz [42], and kAFL [35] use Intel PT [31] to efficiently collect control flow data from the target program. UnTracer [29], instead of tracing every round of execution, instruments the target programs such that the tracing only starts when new code is reached. RetroWrite [11] proposes static binary rewriting to trace code coverage in binary code without heavy dynamic instrumentation.

7.3 Improvements to Parallel Fuzzing

There are two lines of efforts towards better parallel fuzzing in the literature. Xu et al. [46] design new primitives to mitigate the contention in the file system and extend the scalability of the *fork* system call. These new primitives speed up the execution of the target programs when many instances are running in parallel. This line of efforts facilitates parallel fuzzing from a system perspective, which is orthogonal to our approach. Following the other line, P-FUZZ [39], PAFL [22] and Ye et al. [47] propose to distribute fuzzing tasks to different instances to avoid overlaps. We omit the details of P-FUZZ and PAFL since they have been discussed in Sect. 4 and evaluated in Sect. 6. The idea of [47] is to assign seeds that cover less-visited branches to different instances and further confine the mutations to focus on those branches. In comparison to AFL-EDGE, such an idea may skip the exploration of certain code regions and hurt the related fuzzing space. Further, this idea is essentially a variant of PAFL, and thus, we do not compare it with AFL-EDGE in our evaluation.

8 Discussion

In this section, we discuss some of the limitations in our work and the potential future directions.

8.1 Threats to Validity

The validity of our research faces three threats. First, our research is motivated by the intuition that overlapped mutations can reduce the efficiency of code coverage. Whether such an intuition is correct or not threatens the foundation of our research. To mitigate this threat, as presented in Sect. 2, we provide empirical evidence to support the fidelity of our intuition through empirical experiments with real-world programs. Second, AFL-EDGE skips seeds during task distribution, which by theory may reduce the fuzzing space. To validate this threat, we perform extensive experiments to show that AFL-EDGE largely preserves the edge coverage of AFL and thus, avoids hurting the fuzzing space (see Sect. 6.2). Finally, AFL-EDGE and AFL may detect different bugs and AFL-EDGE may miss the bugs detected by AFL. While we provide no theoretical proofs, our empirical evaluation with both real-world programs and standard benchmarks, as shown in Sect. 6.3, argues against such a threat.

8.2 More Fine-Grained Task Distributions Are Needed

AFL-EDGE considers a round of mutations to a seed as an individual task. This represents a coarse-grained definition of fuzzing tasks, which can still result in over-laps. For example, we cannot avoid overlapped mutations by different instances to different seeds. For further improvements, an example idea is to adopt more fine-grained definitions of tasks (e.g., defining fuzzing tasks based on mutations [47]).

8.3 Workloads Need to Be Considered

AFL-EDGE does not explicitly consider the workloads of different tasks. Instead, it relies on random distribution, expecting to achieve probabilistic equivalent workload assignment. This strategy can be further improved by estimating the workload attached to a seed. For instance, we can do such an estimation based on the size of the seed and the execution complexity of the seed (following the idea of AFL-CMIN [50] and QSYM [49]). We may also customize the estimation based on how the fuzzing tools determine the mutation cycles.

9 Conclusion

This paper focuses on the problem of parallel fuzzing. It presents a study to understand the limitations of the parallel mode in the existing grey-box fuzzing tools. Motivated by the study, we propose a general model to describe parallel fuzzing. This model distributes mutually-exclusive yet similarly-weighted tasks to different instances, facilitating concurrency and also fairness across instances. Guided by our model, we present a novel solution to improve the parallel mode in AFL. During fuzzing, our solution periodically distributes seeds that carry non-overlapped and similarly-weighted tasks to different instances, maximally meeting the requirements of our model. We have implemented our solution on top of AFL and we have evaluated our implementation with AFL on 9 widely used benchmark programs. Our evaluation shows that our solution can significantly reduce the overlaps and hence, accelerate the code coverage.

Acknowledgments. We would like to thank the anonymous reviewers for their feedback. This project was supported by NSF (Grant #: CNS-2031377). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

1. Abdelnur, H., State, R., Lucangeli, O.J., Festor, O.: Spectral fuzzing: Evaluation & feedback. [Research Report] RR-7193, NRIA (2010)
2. Aitel, D.: An introduction to spike, the fuzzer creation kit. Proceedings of the Black Hat USA (2002)
3. Anvin, H.P., Gorcunov, C., Chang Seok Bae, J.K., Kotler, F.B.: Nasm source code, July 1996. https://repo.or.cz/w/nasm.git
4. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: Redqueen: Fuzzing with input-to-state correspondence. In: Proceedings of the 2019 Network and Distributed System Security Symposium, vol. 19, pp. 1–15. NDSS, Universitätsstraße 150, 44801 Bochum, Germany (2019)
5. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley, Hoboken (1995)
6. Bellard, F.: Ffmpeg source code. https://ffmpeg.org/releases/ffmpeg-4.1.tar.bz2
7. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1032–1043. ACM, google (2016)

8. Chen, P., Chen, H.: Angora: efficient fuzzing by principled search. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy, pp. 711–725. IEEE, Symposium on Security and Privacy, 1 Shields Ave, Davis, CA 95616 (2018)
9. Chen, Y., et al.: Savior: towards bug-driven hybrid testing. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy, pp. 1–14. IEEE, Symposium on Security and Privacy, San Francisco, CA, USA (2020)
10. Chen, Y., et al.: Ptrix: efficient hardware-assisted fuzzing for cots binary. In: Proceedings of the 2019 ACM on Asia Conference on Computer and Communications Security, pp. 633–645. ACM, AsiaCCS, Auckland, New Zealand (2019)
11. Dinesh, S.: RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. Ph.D. thesis, figshare (2019)
12. Dolan-Gavitt, B., et al.: Lava: large-scale automated vulnerability addition. In: Proceedings of the 2016 IEEE Symposium on Security and Privacy, pp. 110–121. IEEE (2016)
13. Gan, S., et al.: Collafl: path sensitive fuzzing. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy, pp. 660–677. No. 6, Symposium on Security and Privacy, San Francisco, CA, USA, May 2018
14. Gan, S., et al.: GREYONE: data flow sensitive fuzzing. In: Proceedings of the 29th USENIX Security Symposium, pp. 1–18. USENIX Association, Boston, MA, August 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/gan
15. GNU: Index of /gnu/binutils, October 2019. https://ftp.gnu.org/gnu/binutils/
16. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
17. Hayes, L., et al.: Moonlight: Effective fuzzing with near-optimal corpus distillation. arXiv:1905.13055v1 (2019)
18. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138. ACM, CCS, Toronto, ON, Canada (2018)
19. Leffler, S.: Libtiff source code, November 2019. https://download.osgeo.org/libtiff/
20. Lemieux, C., Sen, K.: Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485. ase, Montpellier, France (2018)
21. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 627–637. ACM, fse, PADERBORN, GERMANY (2017)
22. 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, pp. 809–814 (2018)
23. Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R.: Mopt: Optimized mutation scheduling for fuzzers. In: Proceedings of the 28th USENIX Security Symposium, pp. 1949–1966. USENIX, Santa Clara, CA, USA (2019)
24. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Software Engineering, 2007. ICSE 2007. 29th International Conference on. pp. 416–426. IEEE, ICSE, Minneapolis, MN, USA (2007)
25. Manès, V.J.M., et al.: The art, science, and engineering of fuzzing: a survey. IEEE Trans. Softw. Eng. **PP**(21), 21 (2019)

26. Sergeant, M., Christian Glahn, P.P.: Libxml2 source code, September 1999. http://xmlsoft.org/libxml2/libxml2-git-snapshot.tar.gz
27. McKnight, P.E., Najab, J.: Mann-whitney u test. The Corsini encyclopedia of psychology **3**, 960–961 (2010)
28. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, Hoboken (2011)
29. Nagy, S., Hicks, M.: Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy, pp. 787–802. IEEE, Symposium on Security and Privacy, SAN FRANCISCO, CA, USA (2019)
30. Pak, B.S.: Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. School of Computer Science Carnegie Mellon University 0, 1–9 (2012)
31. R., J.: Intel processor trace (2013). https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
32. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium, pp. 1–14. NDSS, San Diego, California, USA (2017)
33. Rebert, A., et al.: Optimizing seed selection for fuzzing. In: Proceedings of the 23rd USENIX Conference on Security Symposium, pp. 861–875. USENIX Association (2014)
34. Scale, F.: Top 10 software testing trends, February 2019. https://fullscale.io/top-10-software-testing-trends/
35. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kafl: hardware-assisted feedback fuzzing for OS kernels. In: Proceedings of the 26th USENIX Conference on Security Symposium, pp. 167–182. USENIX Association, Vancouver, BC, Canada (2017)
36. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. pp. 28–28. USENIX Association, Bellevue, WA, USA (2012)
37. Serebryany, K.: libfuzzer-a library for coverage-guided fuzz testing. LLVM project, p. 1 (2015)
38. She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: Neuzz: efficient fuzzing with neural program smoothing. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy, pp. 1–15. IEEE, Symposium on Security and Privacy, San Francisco, CA, USA (2018)
39. Song, C., Zhou, X., Yin, Q., He, X., Zhang, H., Lu, K.: P-fuzz: a parallel grey-box fuzzing framework. Appl. Sci. **9**(23), 5100 (2019)
40. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: In Proceedings of the 2016 Network and Distributed System Security Symposium, vol. 16, pp. 1–16. NDSS, San Diego, California, USA (2016)
41. McCanne, S., Craig Leres, V.J.: Tcpdump source code, October 2019. http://www.tcpdump.org/release/
42. Swiecki, R.: Honggfuzz (2015). http://honggfuzz.com
43. Teams, T.G.: Oss-fuzz - continuous fuzzing for open source software (2015). https://github.com/google/oss-fuzz
44. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 497–512. IEEE, Oakland, CA, United States (2010)

45. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the Computer and Communications Security 2013. Association for Computing Machinery, New York, NY, USA (2013)
46. Xu, W., Kashyap, S., Min, C., Kim, T.: Designing new operating primitives to improve fuzzing performance. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2313–2328. Association for Computing Machinery, New York, NY, United States (2017)
47. Ye, J., Zhang, B., Li, R., Feng, C., Tang, C.: Program state sensitive parallel fuzzing for real world software. IEEE Access **7**, 42557–42564 (2019)
48. You, W., et al.: Profuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy. IEEE, San Fransisco (2019)
49. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: Proceedings of the 27th USENIX Conference on Security Symposium, pp. 745–761. USENIX Association, Baltimore, MD, USA (2018)
50. Zalewski, M.: afl-cmin, November 2013. https://github.com/mirrorer/afl/blob/master/afl-cmin
51. Zalewski, M., Google: Tips for parallel fuzzing, November 2013. https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt
52. Zalewski, M.: Afl technical details (2013). http://lcamtuf.coredump.cx/afl/technical_details.txt
53. Zhao, L., Duan, Y., Yin, H., Xuan, J.: Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: Proceedings of the 2019 Network and Distributed System Security Symposium, p. 15. NDSS Symposium, San Diego, California (2019)