Dodrio: Parallelizing Taint Analysis Based Fuzzing via Redundancy-Free Scheduling

Jie Liang
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

Mingzhe Wang
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

Chijin Zhou
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

Zhiyong Wu
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

Jianzhong Liu
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

Yu Jiang*
KLISS, BNRist, School of Software, Tsinghua University
Beijing, China

ABSTRACT

Taint analysis significantly enhances the capacity of fuzzing to navigate intricate constraints and delve into the state spaces of the target program. However, practical scenarios involving taint analysis based fuzzers with the common parallel mode still have limitations in terms of overall throughput. These limitations primarily stem from redundant taint analyses and mutations among different fuzzer instances. In this paper, we propose Dodrio, a framework that parallelizes taint analysis based fuzzing. The main idea is to schedule fuzzing tasks in a balanced way by exploiting real-time global state. It consists of two modules: real-time synchronization and load-balanced task dispatch. Real-time synchronization updates global states among all instances by utilizing dual global coverage bitmaps to reduce data race. Based on the global state, load-balanced task dispatch efficiently allocates different tasks to different instances, thereby minimizing redundant behaviors and maximizing the utilization of computing resources.

We evaluated Dodrio on real-world programs both in Google's fuzzer-test-suite and FuzzBench against AFL's classical parallel mode, PAFL, and Ye's PAFL on parallelizing two taint analysis based fuzzer FairFuzz and PATA. The results show that Dodrio achieved an average speedup of 123%–398% in covering basic blocks compared to others. Based on the speedup, Dodrio found 5%–16% more basic blocks. We also assessed the scalability of Dodrio. With the same resources, the coverage improvement increases from 4% to 35% when the number of instances in parallel (i.e., CPU cores) increases from 4 to 64, compared to the classical parallel mode.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Parallel, Fuzzing, Software Testing

ACM Reference Format:
Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Jianzhong Liu, and Yu Jiang. 2024. Dodrio: Parallelizing Taint Analysis Based Fuzzing via Redundancy-Free Scheduling. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3663529.3663844

1 INTRODUCTION

Software vulnerabilities pose significant risks to the security and stability of modern computer systems, requiring effective techniques to detect and mitigate them. Fuzzing has emerged as a powerful automated testing technique that aims to uncover software vulnerabilities by systematically feeding malformed or unexpected inputs into a target program [22, 32, 34, 35, 42]. By observing the program's response to these inputs, fuzzing can identify potential security flaws and areas of weakness [8, 10, 17, 20].

Taint analysis based fuzzing significantly enhances fuzzing by facilitating the navigation of intricate constraints [4, 11, 15, 21]. As software systems become increasingly complex, path constraints often become more intricate. Blind mutation algorithms that are used by conventional fuzzers may generate seeds that get stuck in shallow areas blocked by these constraints, thereby missing potential bugs in deeper logic. By identifying how the input byte influences branches, taint analysis helps fuzzers in mutating bytes that are related to these branches. On the other hand, parallel fuzzing is a common method that is utilized to detect vulnerabilities quickly and early. For large-scale modern software, the need for efficient and scalable vulnerability detection techniques becomes paramount. By distributing the fuzzing workload across multiple instances, parallelization can significantly increase the rate at which vulnerabilities are discovered. The state-of-the-art parallelization approach involves creating multiple instances, where each instance maintains its own seed pool and coverage [8, 18, 25, 38, 41]. These instances are coordinated through periodic seed synchronization.

However, most of the fuzzers utilizing taint analysis seldom consider their performance in common parallel mode. For example, Angora [7] and VUzzer [28] do not support parallel mode. Some taint analysis based fuzzer like FairFuzz [15] support classical parallel mode, but the practice shows that they

*Yu Jiang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663844

even perform worse than the version without taint analysis in parallel mode [18]. In our experiments, using the same resources, the coverage of PATA continuously decreases as the number of cores increases. At 64 cores, the coverage decreases by 22% compared to a single core. *Their performance degradation is mainly due to the redundant behaviors when combining taint analysis and fuzzing in parallel mode.* Traditional parallel fuzzing introduces redundancy as each instance redundantly executes the same seed multiple times to identify new coverage specific to itself. More importantly, seeds are saved repeatedly across instances, leading to duplicate taint analysis and mutations. It is worth noting that taint analyses are usually resource-intensive and have a more deterministic behavior compared to random mutations. Instead of efficiently exploring the new state space, these redundant analyses and mutations divert lots of resources from other potentially fruitful mutations. As a result, the redundancy ultimately results in inefficient utilization of computational resources and diminishes the overall effectiveness of the fuzzing process. In an ideal parallel fuzzing scenario, all fuzzer instances work together as a unified entity, leveraging the full computing power available to minimize redundancies. To efficiently parallelize taint analysis based fuzzing, we need to address the following two challenges:

(1) *Synchronizing the global fuzzing state in real time with limited data races*: in parallel fuzzing, multiple instances execute fuzzing tasks concurrently, which can lead to latency in obtaining global states. Consequently, each instance might utilize outdated information to make fuzzing decisions, leading to suboptimal strategies. Real-time synchronization of this state is necessary to ensure that each instance has access to global information and can make informed fuzzing strategies. However, achieving real-time synchronization requires developing efficient mechanisms to coordinate access to shared resources, such as coverage information and input seeds that cover new basic code blocks. These mechanisms must maintain consistency and prevent race conditions. Striking a balance between synchronization overhead and the accuracy of the global state is crucial to maintaining the effectiveness of the taint analysis based fuzzing in a parallel environment.

(2) *Distributing tasks without duplication or omission*: in parallel fuzzing, distributing tasks across multiple instances or threads is essential to fully utilize the available computing resources and maximize the efficiency of vulnerability detection. However, achieving efficient task distribution without redundancy or omission is challenging. Ensuring that each fuzzing task, such as taint analysis and random mutation, is assigned to an appropriate instance without duplication or omission is crucial for balanced workload distribution. This requires careful load-balancing strategies and task-dispatching mechanisms that consider factors such as the available computational resources and the progress of ongoing taint analysis based fuzzing iterations. Efficient load balancing ensures optimal utilization of resources and maximizes the chances of discovering vulnerabilities across the entire target program.

In this paper, we propose Dodrio, a framework that parallelizes taint analysis based fuzzing via redundancy-free scheduling. The main idea is to schedule fuzzing tasks by utilizing real-time global state, thereby minimizing redundancy and enabling all instances to function as a cohesive unit. The framework incorporates real-time state synchronization and load-balanced task dispatching. Real-time state synchronization ensures that the global state is updated in real-time. It utilizes dual global coverage bitmaps to update coverage, i.e., a fast global bitmap protected by fine-grained locks to coarsely filter seeds, and an accurate bitmap to further sift them. The dual bitmap reduces unnecessary blocking and allows scheduling different tasks into fuzzer instances to cut down on repetitive actions. Furthermore, load-balanced task dispatch distributes different fuzzing tasks across multiple fuzzer instances effectively. By assigning tasks based on the progress of fuzzing and the availability of resources, the load-balanced strategy ensures that each instance receives an appropriate workload without redundant or omitted tasks.

We evaluated Dodrio on programs that both in Google's fuzzer-test-suite [2] and FuzzBench [1] against AFL's classical parallel mode, PAFL [18], and Ye's PAFL [38] on parallelizing two taint-analysis based fuzzer FairFuzz and PATA. The results show that Dodrio demonstrates excellent performance. It achieved an average speedup of 123%–398% in covering basic blocks compared to these parallelization techniques. The speed increase resulted in a 5%–16% increase in the number of basic blocks found by Dodrio. We also assessed the scalability of Dodrio. In the same resource scenario, as the number of instances (i.e., CPU cores) in parallel is increased from 4 to 64, the number of discovered basic blocks by PATA using the AFL classical parallel mode continuously decreases, while Dodrio-PATA is able to maintain a relatively stable count. Correspondingly, relative to PATA, the number of covered basic blocks by Dodrio-PATA has increased from 4% to 35%.

2 BACKGROUND AND MOTIVATION

Fuzzing. Fuzzing uncovers vulnerabilities in target programs by feeding malformed inputs into them. The inputs are always called seeds in fuzzing. Generally, fuzzers mutate existing seeds to generate new ones. The process always starts from a set of initial seeds, with the feedback guided (e.g. coverage), only the cases that could hit new behaviors (e.g. covering new branches) will be preserved for further mutation. The mutation process typically involves deterministic and random methods. For instance, AFL employs bitflip for each input bit during the deterministic stage and randomly changes byte values during the havoc stage.

Classic Parallel Mode of Fuzzing. A fuzzer could run in single-core mode or parallel mode. In single-core mode, only one fuzzing instance is used to test the target program, while in parallel mode, multiple fuzzing instances are simultaneously employed. The parallel deployment of fuzzing is a common practice in the industrial testing of real-world programs. Many widely used industrial-grade fuzzing tools, such as AFL [41] and Honggfuzz [14], support parallel mode. *Most of these tools employ the classic parallel mode, where different fuzzing instances coordinate their activities by synchronizing the test case corpus.* In classic parallel mode, different fuzzing instances collectively test the same target program. Each instance maintains its own seeds and periodically scans the seeds of others to synchronize useful inputs into its own corpus. By synchronizing the seeds, a fuzzing instance can leverage the state space explored by other instances. The time taken by other instances to reach the same state is reduced, thereby improving the overall efficiency.

Taint Analysis Based Fuzzing. Traditionally, taint analysis traces the propagation of tainted data at a fine-grained level within the

program, marking data that originate from untrusted sources and tracking its flow through various operations and variables. Integrating taint analysis and fuzzing represents a powerful approach to improving the efficiency and effectiveness of fuzzing. Specifically, the combination of taint analysis and fuzzing enables fuzzers to gain insight into the data flow and identify crucial input bytes that influence specific program branches. For example, FairFuzz [15] adopts taint analysis to deduce the relationship between input bytes, rare branches, and mutation operators. It performs byte-level mutation on the input to identify mutation operators for each byte that ensure the mutated seed continues to cover rare branches. PATA [21] utilizes path-aware taint analysis. It takes into account the different paths that tainted data can follow through conditional branches and loops, allowing for more precise identification of vulnerabilities. On the other hand, taint analysis can be computationally expensive and may slow down the fuzzing process. It involves tracking the flow of data through the program and identifying tainted values, which requires additional overhead and computational resources.

Motivation. Ideally, running N fuzzers in parallel should have the same effect as increasing the CPU main frequency by N times when all processes of the fuzzing can be parallelized [3]. Alternatively, parallel fuzzing with N cores for T hours should yield similar results as using T cores for N hours. However, in practice, we find that when using the classical parallel mode, taint analysis based fuzzers may experience a significant performance decline when running on multiple cores. For example, PATA utilizes the classical parallel mode of AFL. Figure 1 shows the trend of the total number of basic blocks covered on tested projects (see Section 5.1) by PATA with the same resources (24 $core * hours$) as cores increase. It shows that the number of basic blocks discovered by PATA decreases continuously as the number of cores increases. Particularly, at 64 cores, the coverage decreases by 22% compared to a single core.

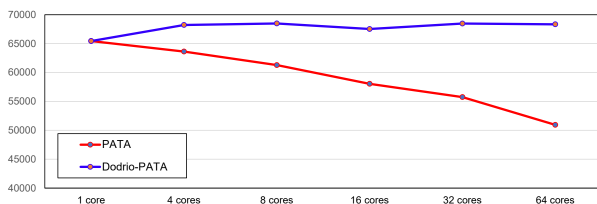

Figure 1: Trend of basic blocks covered on all test projects by PATA and Dodrio-PATA with the same resources (24 core∗hours) as cores increase.

The performance degradation is mainly caused by redundant behaviors of the classical parallel mode. First, one seed might be repeatedly synchronized to all instances, leading to a redundant process that involves re-execution and storage of the same seeds. Moreover, when combining taint analysis with fuzzing, the same seeds repeatedly saved in each instance will be analyzed multiple times, resulting in redundant taint analysis processes across instances. The analysis slows down the whole system's fuzzing speed. Finally, taint analysis based fuzzers mutate input seeds guided by the analysis results. In parallel fuzzing, the similar seeds and analysis would result in redundant similar mutations, consequently, producing alike mutated seeds that have the same contributions to testing. To reduce redundant actions across instances in parallel fuzzing, Dodrio employs real-time state synchronization and load-balanced task dispatch. Figure 1 shows that after when PATA is augmented with Dodrio, there is no significant decrease in the number of covered basic blocks as CPU cores increase.

3 DODRIO DESIGN

Figure 2 presents the overall design of Dodrio. Dodrio has two modules, namely real-time state synchronization and load-balanced task dispatch. All instances share a global coverage bitmap, a seed cache queue, and a seed pool. Dodrio utilizes the latest global coverage bitmap to identify seeds with new coverage, pushing them into the queue and further filtering them into the pool. Moreover, Dodrio dispatches distinct tasks to different instances, which avoids redundant analysis and mutations among fuzzer instances.

3.1 Real-Time State Synchronization

In parallel fuzzing, multiple instances execute fuzzing tasks concurrently, which can introduce latency in obtaining global states. Consequently, each instance might utilize outdated information to make fuzzing decisions, leading to suboptimal strategies from a global perspective. Over time, the accumulation of these suboptimal strategies can significantly diminish the overall effectiveness of fuzzing. Consequently, real-time state synchronization is necessary to enhance the efficiency of parallel fuzzing.

State Synchronization Architecture. Seed corpus and coverage bitmap are the two primary states for fuzzing. The seed corpus represents the discoveries made during the fuzzing process. It is essential because fuzzers take the seeds in the corpus and mutate them to generate new test cases. The coverage bitmap is the data structure that indicates which parts of the code have been executed during fuzzing. The bitmap plays a critical role in guiding mutation-based fuzzers by determining whether a seed can identify uncovered code and need to save for further mutation. It is also the basis for prioritizing seeds for mutation. To obtain the latest state in real-time, an intuitive idea is to utilize the architecture that only has one coverage bitmap and a seed pool across all fuzzer instances. In this way, any change in bitmap caused by new coverage discovered by one instance can be immediately perceived by other fuzzer instances. The challenge with this architecture lies in handling race conditions in access/update seed pool and read/update bitmap.

To address that, as Figure 2 shows, we design the synchronization architecture that has a global seed cache queue, a global seed pool, and dual coverage bitmaps. We use a global lock to protect the seed pool, since the occurrence of data races is limited, given that the frequency of new seed discoveries is not high. To further mitigate competition, we design a seed cache queue that can be used to smooth the updates to the seed pool using task dispatch (see the next section). However, concerning the global bitmap, each generated seed requires reading it to verify if new seeds have been found, which can result in frequent data races. Protecting the entire bitmap with a single lock may lead to a significant decrease in overall efficiency when the number of instances increases, as it can result in frequent waits and contention for the lock.

The main issue is the unnecessary blocking when only using one lock. In practice, most of the newly found basic blocks by different

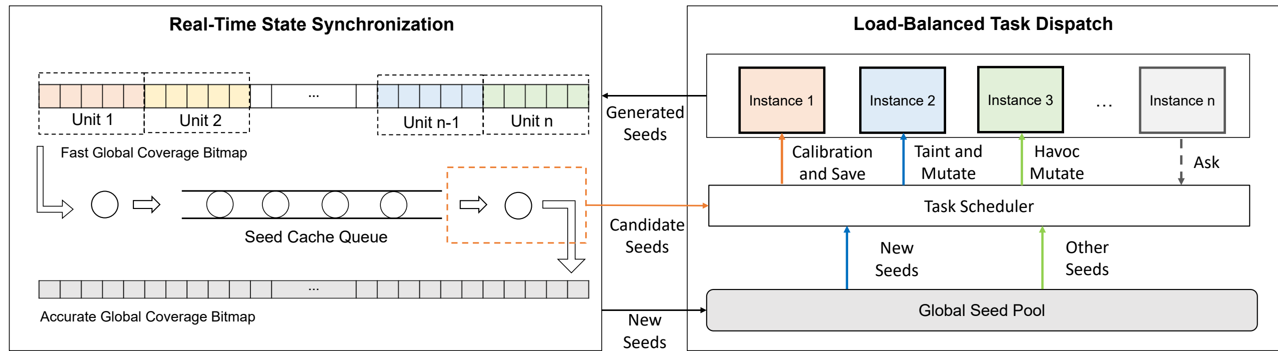

Figure 2: Dodrio parallelizes taint analysis based fuzzing via redundancy-free scheduling. The framework has two modules, namely real-time state synchronization and load-balanced task dispatch. First, it utilizes dual coverage global bitmaps for real-time state synchronization. Correspondingly, it has a global seed cache queue and a global seed pool. The fast global bitmap, protected by fine-grained locks, is used to coarsely filter generated seeds to the cache queue, and an accurate bitmap is used to further sift them to the pool. Moreover, it uses load-balanced task dispatch to distribute different fuzzing tasks across multiple fuzzer instances. When idle instances request tasks, it dispatches saving tasks if there are candidate seeds in the cache queue, or taint tasks if there are new seeds in the global seed pool. If neither condition is met, it dispatches havoc tasks.

instances at the same time are different. In addition, because the blocks are randomly mapped into the bitmap, they are distributed in different areas of the bitmap. Consequently, only using one lock to guard the entire bitmap is unnecessary. In practice, two patterns can be observed in the use of bitmaps by fuzzing: (1) new basic blocks found by different instances at the same time are scattered over different regions of the bitmap; (2) a relatively small percentage of the large number of seeds generated can cover new basic blocks, so it reads the covered bitmaps very frequently, while the write operations are not very frequent. In conjunction with the seed cache queue and global seed pool, we design dual coverage bitmaps to alleviate data contention.

Dual Coverage Bitmaps. The dual coverage bitmap has a *fast coverage bitmap* and an *accurate coverage bitmap*. Dodrio first rapidly determines whether a seed has new coverage using the fast coverage bitmap and pushes the seeds with new coverage into the seed cache queue. Subsequently, the relatively few seeds in the queue are further filtered by the accurate coverage bitmap. The seeds that pass the filtering will be stored in the global seed pool.

The fast coverage bitmap utilizes fine-grained locks. We designate each fine-grained lock-protected region as a "unit". The fine-grained locks protect against the data races for each unit by exploiting atomic operations. When one instance needs to update some units while others need to access other units, they will have no access conflicts. Even if they need to access a common region, the lock will keep the process running smoothly. One problem here is specifying memory orderings[1], namely the way atomic operations synchronize memory. Here we mainly use the relaxed order, namely not imposing an order among concurrent memory accesses. Using strict memory orderings ensures sequential consistency, but it may cost lots of time. In fuzzing, what matters is not leaving out any seeds which results in new coverage. The bitmap is used to detect the seeds which find new coverage. If a store operation happens before a load operation, the bitmap could be updated successfully without saving extra seeds. Otherwise, if a load operation happens before a store operation, the bitmap could still be updated successfully. Because we employ fine-grained locks in the fast coverage bitmap, two seeds that find the same new blocks may both be preserved. The reason is that newly found blocks may be distributed in different areas, which are protected by different locks in the bitmap. The order and the delay of storing value may cause different fuzzer instances to think they find different new coverage in different areas. To eliminate the side effects, we also design an accurate global coverage bitmap. It employs the traditional design, namely synchronizing based on a global lock. With the fast global coverage bitmap, Dodrio first quickly filters out seeds with new code coverage from the large number of seeds generated and stores them in the seed cache queue. Because the number of seeds in the queue will be relatively few, using one global lock is acceptable.

3.2 Load-Balanced Task Dispatch

Load-balanced task dispatch involves effectively distributing fuzzing tasks among a cluster of backend fuzzer instances. In parallel fuzzing, load balancing should not only ensure timely task allocation to each fuzzer instance to avoid idle waiting but also ensure that each instance performs unique tasks.

To ensure prompt task allocation, we employ a proactive approach. When a fuzzer instance is free, it will actively request and retrieve a task for execution. For parallel fuzzing that employs taint analysis, there are mainly three kinds of tasks: ① Multiple executions to calibrate seed information (e.g., coverage bitmap) and save seeds to the global seed pool (referred to as *calibration and save* task). ② Taint analysis and corresponding mutations (referred to as *taint analysis and mutation* task). ③ Random mutations using traditional fuzzing techniques (referred to as *havoc mutation* task). In tasks 2 and 3, an instance will immediately execute the seed it mutated to detect if it has achieved new coverage. Once the task is complete, the instance will proceed to request a new one.

[1]https://en.cppreference.com/w/cpp/atomic/memory_order

The tasks are determined based on the status of the seed cache queue and seed pool. Algorithm 1 illustrates the process of task dispatch. A fuzzer instance will continuously retrieve tasks based on the global seed pool and the cache queue, and then execute the task (Lines 1-4). In retrieving a task, when the seed cache queue is not empty, the fuzzing instance retrieves the seed at the front of the queue and determines whether it can overwrite the new basic block based on the global coverage bitmap (Lines 6-9). If true, the algorithm generates and assigns a *calibration and saving task* (Lines 10-12). If not, the algorithm attempts the next seed in the queue until the queue is empty.

If the cache queue is empty, the algorithm randomly selects a seed from the global seed pool for mutation. If this seed has not undergone mutation with taint analysis, the algorithm generates and assigns a *taint analysis and mutation task* (Lines 16-19). This task aims to infer the taints with path-aware methods within the seed and perform mutations accordingly, exploring potential vulnerabilities or coverage paths. If taint analysis has already been performed, the algorithm generates and assigns a *havoc mutation task*. The havoc mutation task follows the algorithm of AFL [41] and requires specifying the number of mutations based on the seed's execution information. Additionally, it needs to prepare other seeds for cross-mutation from the seed pool.

Algorithm 1: Load-Balanced Task Dispatching

```
    Input  : Seed queue: queue,
             Global seed pool: pool,
             Accurate coverage bitmap: accurate_cov
 1  while True do
 2  |   task = retriveTask(queue, pool, accuracte_cov);
 3  |   executeTask(task);
 4  end
 5  Function retriveTask(queue, pool, accuracte_cov):
 6  |   seed = popFront(queue);
 7  |   while seed ≠ NULL do
 8  |   |   cov = run(seed);
 9  |   |   if hasNew(cov, accurate_cov) then
10  |   |   |   task = calibrateAndSave(seed, accurate_cov);
11  |   |   |   return task;
12  |   |   end
13  |   |   seed = popFront(queue);
14  |   end
15  |   seed=select(pool);
16  |   if hasNotTaintMutated(seed) then
17  |   |   task = taintAndMutate(seed);
18  |   |   return task;
19  |   end
20  |   task = havocMutate(seed, pool);
21  |   return task;
22  End Function
```

4 IMPLEMENTATION

Dodrio is implemented in Rust. As Figure 2 shows, it has mainly two modules, namely real-time state synchronizer and load-balanced task dispatcher.

The real-time state synchronizer maintains a fast coverage bitmap and an accurate bitmap. Both bitmaps are stored in shared memory to support zero-copy communication. The fast bitmap is implemented as an array of atomic 64-bit (i.e., the unit length in Figure 2) unsigned integers, which is shared among all instances of the fuzzer. Each fuzzer instance is equipped with a fast coverage updater responsible for reading and updating the fast coverage bitmap. These updates employ relaxed memory orderings when loading and storing values. Whenever new coverage is detected, the corresponding candidate seed is added to the seed cache queue. Furthermore, the accurate bitmap is implemented as an array of 8-bit integers, with the same size as the fast bitmap. To ensure thread safety and consistent updates, it is protected by an atomic reference counter and a mutex. The atomic reference counter guarantees that concurrent access to the accurate bitmap is synchronized and prevents data races. The mutex provides exclusive access to the accurate bitmap, allowing only one instance to modify it at a time. The accurate bitmap is updated by an accurate bitmap updater within each fuzzer instance. Besides, the real-time state synchronizer also maintains a global seed pool and a seed cache queue. They are also protected by atomic reference counters and mutexes.

The task scheduler dispatches tasks according to Algorithm 1. The calibration and save task, havoc mutation task, and seed selection follow the algorithm of AFL. We also implement the fork server of AFL for seed execution. Dodrio supports packaging taint analysis and mutation tasks as a plugin in a separate dynamic shared object (DSO) file, which can be dynamically loaded and linked to the main program at runtime. It requires the implementation of basic predefined interfaces, such as initialization, analysis, and mutation. Adapting a new fuzzer only requires converting its mutation methods to such interfaces. Currently, we have implemented parallelization for FairFuzz and PATA.

5 EVALUATION

We evaluated Dodrio in terms of its ability to scale taint-analysis fuzzers to parallel mode. Our evaluation aims to answer the following questions:

- **RQ1:** How does Dodrio perform compared to other parallelization techniques?
- **RQ2:** What is the contribution of Dodrio's modules?
- **RQ3:** How is the scalability of Dodrio with respect to increasing CPU cores?

5.1 Evaluation Setup

Experiment Environment. The overall experiments were conducted on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 504 GiB of main memory. We maintained identical configurations for each fuzzer and target application. Each fuzzer was executed with 4 CPU cores continuously for a duration of 24 hours. The binaries for all fuzzers were built with AddressSanitizer (ASAN) [29] enabled.

Benchmark and Initial Seeds. We perform evaluation on programs that exist in both widely-used benchmarks: Google's fuzzer-test-suite [2] and FuzzBench [1]. The target programs are carefully picked by Google, which consists of a comprehensive set of popular real-world programs. For the target programs, all fuzzers used the

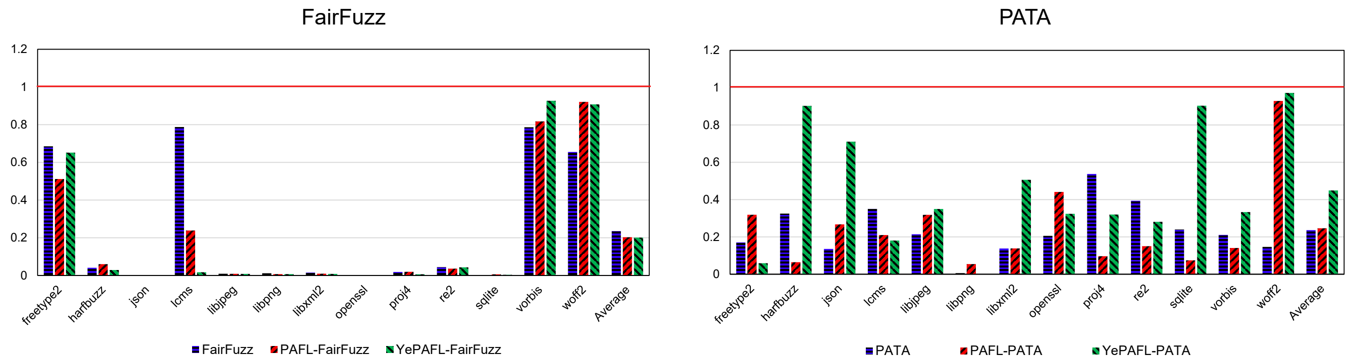

Figure 3: Normalized execution time required by Dodrio to cover the same basic blocks as other parallelization methods on FairFuzz (left) and PATA (right) in 24 hours. The X-axis is the target programs, and the Y-axis is the ratio between the execution times required for achieving the same coverage (i.e., the maximum coverage achieved by the other technologies). A bar below the red line indicates a speed improvement in covering basic blocks for Dodrio than other parallelization methods.

same seeds collected from the data set. In cases where there were no available seeds, an empty seed was used as a fallback option.

Performance Metrics. During the evaluation, we assessed the performance of the fuzzers using three key metrics: the speed in covering basic blocks, the number of basic blocks, and the number of triggered bugs. To ensure a fair and consistent comparison among different fuzzers that may have distinct representations of fuzzing states, we adopted a unified approach to identify basic blocks. Specifically, we collected and executed the seeds generated by each fuzzer during the evaluation. We utilized LLVM tools to identify basic blocks to count the unified number.

Another important aspect we considered during the evaluation was the number of bugs triggered by each fuzzer. However, different fuzzers employ various methods to distinguish unique crashes, which can affect the reported numbers. To ensure a fair and accurate comparison, we followed a two-step process. Firstly, we collected the crash inputs that were triggered during the fuzzing process. These inputs were then re-executed, and the call stack was backtracked to filter out any redundant or overlapping crashes. Moreover, to enhance the accuracy of bug identification, we conducted a manual analysis of the identified bugs. This manual analysis allowed us to eliminate duplicate entries and ensure that each reported bug was unique and distinct.

5.2 Comparison With Existing Parallelizations

To evaluate the effectiveness of redundancy-free parallelization, we compare Dodrio against the classical parallel mode of AFL [41], and the two state-of-the-art parallelization works, namely PAFL [18] and YePAFL[2] [38]. For the parallel mode of AFL, each instance manages its own seed pool and regularly synchronizes the seeds from the pools of other instances. PAFL regularly synchronizes global and local guiding information and dispatches fuzzing tasks by dividing bitmap statically. YePAFL is also an advanced parallel fuzzing work, which optimizes parallel fuzzing by designing global coverage bitmap and on-demand synchronization. To demonstrate the effectiveness of these techniques on parallelizing taint based fuzzers, we applied them to parallelize FairFuzz [15] and PATA [21]. FairFuzz leverages byte-level taint analysis to deduce mutation operators for each byte, ensuring the preservation of rare branch reachability. PATA employs path-aware taint analysis to identify critical bytes for each access of a constraint. Specifically, for FairFuzz, we conducted a comparative analysis between Dodrio-FairFuzz, FairFuzz, PAFL-FairFuzz, and YePAFL-FairFuzz. For PATA, we compare Dodrio-PATA against PATA, PAFL-PATA, and YePAFL-PATA. Since some of these works are not available, we re-implemented these parallelization works based on the papers describing these techniques. The evaluation was performed on target projects with fuzzing 24 hours on 4 CPU cores.

Speed in Covering Basic Blocks. Figure 3 shows the comparison of time required by Dodrio to reach the same coverage as other parallelization techniques. The left and right sides represent the speedup of Dodrio on the parallelizations of FairFuzz and PATA, respectively, compared to other parallelization methods. The blue, red, and green bars represent the parallelization of Dodrio in comparison to other methods to parallelize taint analysis based fuzzers, including the AFL classical parallel mode, the parallelization of PAFL, and the parallelization of YePAFL, respectively. The Y-axis of the graph represents the ratio of the time for Dodrio to reach the same coverage (i.e., the maximum coverage achieved by the other technologies) to the time for the other technologies. A bar below the red line indicates a speed improvement in covering basic blocks for Dodrio than other parallelization methods in experiments.

The left side of the figure illustrates that, on average, Dodrio-FairFuzz achieved a time ratio of 0.24, 0.20, and 0.20 compared to FairFuzz, PAFL-FairFuzz, and YePAFL-FairFuzz, respectively, when applied to FairFuzz. *It means Dodrio-FairFuzz achieved an average speedup of 325%, 393%, and 398% in covering basic blocks relative to the parallelization of FairFuzz, PAFL-FairFuzz, and YePAFL-FairFuzz, respectively.* In other words, Dodrio takes only 5.64, 4.86, and 4.82 hours, respectively, to cover the maximum coverage that other technologies can achieve in 24 hours. Similarly, the right side of the graph demonstrates that, on average, Dodrio achieved a

[2]It is also called PAFL in its paper. To make a better distinction, we use YePAFL to denote it.

Table 1: Number of basic blocks covered by different parallelization technologies on FairFuzz and PATA in 24 hours

Project	FairFuzz	PAFL-FairFuzz	YePAFL-FairFuzz	Dodrio-FairFuzz	PATA	PAFL-PATA	YePAFL-PATA	Dodrio-PATA
freetype2	12718	12682	12735	13017	16033	16351	14511	16721
harfbuzz	9498	9698	9320	10332	10172	10003	10557	10596
json	1579	1564	1583	1603	1603	1601	1603	1603
lcms	1649	1653	1649	1660	2614	2701	2698	2810
libjpeg	2369	2458	2375	2484	2826	2838	2840	2859
libpng	1191	1182	1172	1203	1371	1352	1373	1432
libxml2	6118	4754	4624	9562	9562	9552	9921	11160
openssl	5305	5304	5304	5571	5569	5569	5589	5656
proj4	961	941	828	2157	5393	5335	5357	5398
re2	5632	5613	5624	5717	5722	5711	5717	5770
sqlite	1805	1805	1805	1808	1808	1808	1808	1808
vorbis	2042	2050	2050	2063	2102	2092	2104	2121
woff2	3034	3011	3006	3050	2966	3117	3003	3168
Total	53901	52715	52075	60227	67741	68030	67081	71102
Improvement	12%↑	14%↑	16%↑	–	5%↑	5 %↑	6%↑	–

time ratio of 0.24, 0.25, and 0.45 compared to the parallelization of AFL, PAFL, and YePAFL, respectively, when applied to PATA. *This translates to Dodrio achieving an average speedup of 323%, 306%, and 123% in terms of basic block coverage relative to the parallelization of PATA (i.e., AFL classical parallel mode), PAFL, and YePAFL, respectively.* In other words, Dodrio-PATA requires 5.68, 5.92, and 10.79 hours to achieve the maximum coverage of PATA, PAFL-PATA, YePAFL-PATA in 24 hours, respectively.

To investigate the speed-up in the whole process of parallel fuzzing, we use Figure 4 to demonstrate the average speed-up ratio in 3, 6, 12, and 24 hours with respect to classical AFL parallel mode, PAFL's parallelization, and YePAFL's parallelization, respectively. It illustrates that Dodrio consistently outperformed other technologies throughout the entire process. The bars representing 3, 6, and 12 hours indicate the average speedup observed during shorter experiments. During these time intervals, fuzzing is typically not fully saturated. Fluctuations in the acceleration ratios can be observed within these phases due to random factors, such as the selection of seeds for mutation, which can influence acceleration. Nevertheless, Dodrio still manages to achieve a speedup in covering basic blocks.

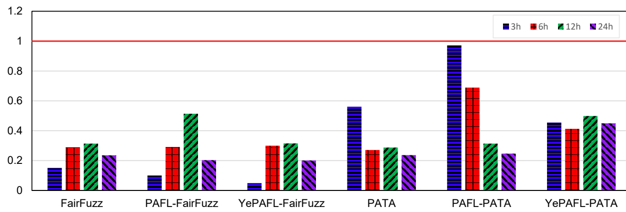

Figure 4: Normalized execution time required by Dodrio to reach the same coverage as other methods in 3, 6, 12, and 24 hours. The X-axis is parallelization methods, and the Y-axis is the average ratio between the execution times required for achieving the same coverage. A bar below the red line indicates a speed improvement for Dodrio than others.

Since the taint analysis based fuzzing algorithms used for the different parallelization technologies are the same, the speedup is mainly due to the reduction of redundant behaviors by the parallelization of Dodrio. PAFL and YePAFL improve parallel fuzzing by synchronizing coverage information with a global bitmap and local bitmaps in each instance. Nevertheless, their parallel mode still follows the common methods of AFL, namely, maintaining seed corpus for each instance and synchronizing seeds regularly. The seed synchronization consumes additional resources, and its latency may prevent a single instance from selecting the latest seed, resulting in a decrease in efficiency. Moreover, since each instance selects its seed for mutation separately, different instances may select the same seed for taint analysis or mutation, which results in duplication and waste of resources.

Differently, Dodrio utilizes real-time synchronization, which only contains one global seed pool. Real-time synchronization allows individual instances to get the global state in real time. Based on that, Dodrio dispatches diverse tasks to different fuzzers according to the global state. Since there is only one seed pool, redundant behavior can be reduced by accurately assigning different tasks to different fuzzing instances in the global state. As a result, the parallel mode of the taint analysis based fuzzing gains speed in covering the basic blocks, and the overall efficiency is improved.

Basic Blocks. Table 1 shows the number of blocks covered by each parallelization method. Dodrio-FairFuzz covered 12%, 14%, and 16% more basic blocks compared to FairFuzz, PAFL-FairFuzz, and YePAFL-FairFuzz respectively. Also, Dodrio-PATA covered 5%, 5%, and 6% more basic blocks compared to PATA, PAFL-PATA, and YePAFL-PATA respectively. The improvement is consistent across individual programs, as Dodrio covered the highest number of program basic blocks, whether for FairFuzz or PATA.

Despite no changes to the algorithms of the fuzzers themselves, Dodrio manages to drive FairFuzz and PATA to cover more basic

Table 2: Number of bugs triggered by different parallelization technologies on FairFuzz and PATA in 24 hours

Project	FairFuzz	PAFL-FairFuzz	YePAFL-FairFuzz	Dodrio-FairFuzz	PATA	PAFL-PATA	YePAFL-PATA	Dodrio-PATA
harfbuzz	0	0	0	0	0	0	0	1
json	1	1	1	2	2	2	1	2
lcms	1	1	1	1	1	1	1	1
libxml2	1	2	1	2	2	2	1	4
re2	0	0	0	1	1	1	1	2
woff2	1	1	1	1	1	1	1	1
Total	4	5	4	7	7	7	5	11

blocks in parallel mode. This is mainly due to the increased speed in covering basic blocks of Dodrio. By covering basic blocks faster, Dodrio enables the underlying taint analysis based fuzzing methods to focus on uncovered new logical areas of code earlier, and thus achieve higher coverage.

It is noted that some projects like `sqlite` show only minimal improvement. This is because the fuzzing algorithm has reached saturation during the 24-hour parallel experiment, and the number of covered basic blocks cannot be further increased by simply increasing the effective execution iterations. The table also shows that the improvement in covered basic blocks on FairFuzz is higher than PATA. It is because the algorithm of FairFuzz has higher randomness than PATA, requiring more effective execution iterations to achieve the condition of covering basic blocks. The requirement for more effective execution iterations provides Dodrio with a greater improvement potential in terms of covering basic blocks. Moreover, within 24 hours, the method of PATA may reach saturation and has found most of the basic blocks it can cover.

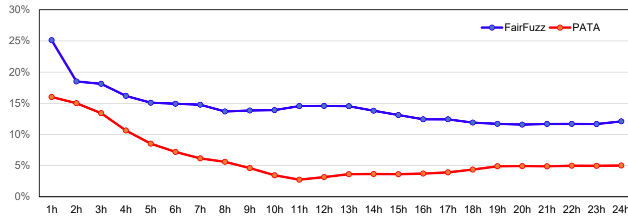

Figure 5: Coverage improvement by Dodrio on FairFuzz and PATA within 24 hours.

Nevertheless, during the early stages of the testing process when the algorithm has not yet reached saturation, Dodrio can still achieve greater improvements in coverage of basic blocks. Figure 5 shows the coverage improvement by Dodrio on FairFuzz and PATA within 24 hours. It shows that in the first 4 hours, Dodrio has achieved coverage improvements of over 10% compared to using the classical parallel mode of PATA.

Bug Triggering. Table 2 presents the number of bugs triggered by each parallelization technology on FairFuzz and PATA. The identification of bugs involves a comparison of the call stack and the execution of manual analysis to differentiate them. According to the table, FairFuzz, PAFL-FairFuzz, YePAFL-FairFuzz, Dodrio-FairFuzz, PATA, PAFL-PATA, YePAFL-PATA, and Dodrio-PATA triggered 4, 5, 4, 7, 7, 7, 5, and 11 bugs, respectively.

The improvement in speed and coverage increases Dodrio's possibility to help FairFuzz and Dodrio of finding bugs. Dodrio-FairFuzz found more bugs than PAFL-FairFuzz and YePAFL-FairFuzz because it finds more basic blocks in 24 hours and it increases the possibility of triggering logic in deep states. With path-aware taint analysis, PATA and most of its parallelization found more bugs than FairFuzz series. YePAFL tries to reduce synchronization overhead. Its master node synchronizes the seeds from each slave node, and the slave nodes only synchronize from the master node. However, this approach exacerbates latency and YePAFL-PATA finds fewer bugs than PATA. By reducing redundancy and increasing the effective coverage of program code logic, Dodrio enhances the coverage of basic blocks, enabling it to discover more bugs compared to other parallelization methods.

5.3 Contribution of Modules

To evaluate each module, we implemented two variations of Dodrio-PATA: Dodrio-disreal and Dodrio-distask. Dodrio-disreal disables real-time state synchronization, which relies on fast and accurate coverage bitmaps, and instead utilizes a single global bitmap along with a lock mechanism to prevent data races. Dodrio-distask disables load-balanced task dispatching, which assigns tasks based on the global state, and instead employs random dispatching tasks. Note that tasks may be duplicated in different instances. For simplicity, we use Dodrio to represent Dodrio-PATA in this subsection.

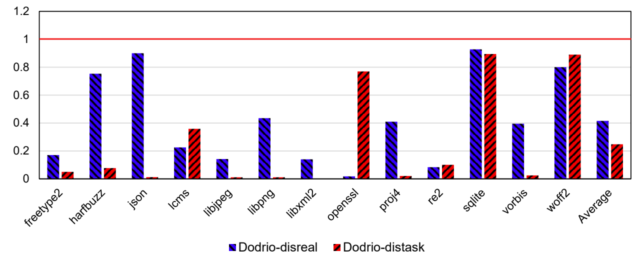

Figure 6: The execution time required by Dodrio to reach the same coverage as Dodrio-disreal and Dodrio-distask in 24 hours. The X-axis is the target programs, and the Y-axis is the ratio between the execution times required for achieving the same coverage. A bar below the red line indicates a speed improvement for Dodrio than others.

Figure 6 shows the execution time required by Dodrio to reach the same coverage as Dodrio-disreal and Dodrio-distask in 24 hours. In the figure, a bar below the red line indicates a speed improvement for Dodrio than others. On average, Dodrio requires 9.97 and 5.95 hours to cover the same number of basic blocks covered by Dodrio-disreal and Dodrio-distask in 24 hours with 4 cores. In other words, Dodrio improves Dodrio-disreal and Dodrio-distask about 141% and 303% in speed of covering basic blocks, respectively. The results indicate that both modules of Dodrio significantly contribute to accelerating the discovery of basic blocks during fuzzing process.

Table 3: Number of basic blocks covered by Dodrio-disreal, Dodrio-distask, and Dodrio in 24 hours

Project	Dodrio-disreal	Dodrio-distask	Dodrio
freetype2	16073	13667	16721
harfbuzz	10344	4834	10596
json	1603	1603	1603
lcms	2599	2453	2810
libjpeg	2815	2400	2859
libpng	1412	1410	1432
libxml2	9552	5318	11160
openssl	5569	5278	5656
proj4	5349	5209	5398
re2	5674	5680	5770
sqlite	1808	1808	1808
vorbis	2104	2034	2121
woff2	3100	2481	3168
Total	68002	54175	71102
Improvement	5%↑	31%↑	–

Table 3 shows the number of basic blocks covered by Dodrio-disreal, Dodrio-distask, and Dodrio in 24 hours. It illustrates that Dodrio finds 5% and 31% more basic blocks than Dodrio-disreal and Dodrio-distask, respectively. It is worth noting that Dodrio-disreal covers more basic blocks than Dodrio-distask due to the reduced redundant behavior. Although Dodrio-disreal introduces some wait time by using only one global lock to update the global coverage bitmap, the task dispatching module still minimizes its redundant behavior in parallel mode. The results show that both real-time state synchronization and load-balanced task dispatching are important for the effectiveness of Dodrio.

5.4 Scalability of Dodrio

To evaluate the scalability of redundancy-free parallelization, we compared Dodrio-PATA against PATA (using the classical parallel model) on tested programs with 4, 8, 16, 32, and 64 cores, for a total of 24 $core * hours$. Table 4 shows the improvements of basic blocks found by Dodrio-PATA compared to PATA. It indicates that when using the same resources, Dodrio-PATA outperforms PATA in terms of the number of covered basic blocks using 4, 8, 16, 32, and 64 CPU cores, with respective increases of 4%, 8%, 12%, 17%, and 35% on average. The results indicate that as the number of CPU cores increases, the improvement in the number of covered basic blocks also increases when compared to PATA. From the table, We can also find that the improvements on some projects are higher than others. We investigate them and find that they execute test cases rather slowly and will cost more time for synchronization.

Table 4: Improvement in the number of covered basic blocks for Dodrio-PATA compared to PATA with 4, 8, 16, 32, and 64 cores for a total of 24 core∗hours

Project	4	8	16	32	64
freetype2	29%	37%	56%	79%	84%
harfbuzz	2%	11%	27%	34%	31%
json	0%	0%	0%	0%	0%
lcms	13%	36%	39%	49%	181%
libjpeg	0%	0%	13%	14%	8%
libpng	3%	3%	3%	3%	17%
libxml2	2%	11%	11%	11%	47%
openssl	0%	0%	0%	0%	0%
proj4	0%	2%	5%	5%	50%
re2	1%	1%	1%	0%	2%
sqlite	0%	0%	0%	0%	0%
vorbis	1%	1%	0%	27%	26%
woff2	0%	1%	2%	3%	5%
Average	4%	8%	12%	17%	35%

Figure 1 in Section 2 shows the trend of the total number of basic blocks covered on tested projects by PATA with the same resources (24 $core * hours$) as cores increase from 1 to 64. It demonstrates that Dodrio is able to maintain a stable number of discovered basic blocks as the number of CPU cores increases. When adding more cores, the classical parallel fuzzing may waste resources because of the repeated saved seeds and repeated actions. Differently, Dodrio synchronizes the global fuzzing states in real-time with limited data race to avoid repeated saving seeds. Furthermore, through load-balanced task dispatching, Dodrio intelligently assigns appropriate tasks to different fuzzer instances, preventing redundant mutations or analyses from occurring. As a result, when adding more cores, Dodrio can make greater improvements in coverage for finding basic blocks compared to the classical parallel mode.

6 LESSONS LEARNED

In this section, we introduce some lessons learned on parallelizing taint analysis based fuzzing.

Decreasing redundant behavior is a practical way to improve the efficiency of parallel fuzzing. The redundant behaviors mainly manifested in the repetitive analysis and mutation of the same seeds in different instances. As a result, many instances will repeatedly produce the same outputs, and the resources are wasted. Reducing redundant behaviors helps different instances achieve varied coverage, thereby enhancing overall effectiveness.

Distributing tasks in a global perspective aids in reducing redundant behavior. Distributing tasks based on a global view (e.g., global seed pool and bitmap coverage) ensures that different instances perform different behaviors, thus the redundant behavior can be reduced. In addition to task assignment, there are other ways to reduce redundant behavior, such as assigning different initial seeds to different fuzzy test instances. However, this approach can fail quickly due to seed synchronization.

The parallelization design should consider the compatibility with the technology (e.g., taint analysis) used in a fuzzer. The parallelization of DODRIO does not affect the internal taint analysis algorithms. By utilizing task dispatch, the taint analysis could be conducted in a load-balanced way. Besides taint analysis, redundancy-free scheduling could also be utilized to parallelize other resource-intensive analysis techniques.

7 RELATED WORK

Mutation-Based Fuzzing. Mutation-based fuzzing typically encompasses three steps: coverage instrumentation, seed selection, and seed mutation. AFL [41] is a popular fuzzer. It instruments code to track branch coverage and implements essential selection and mutation operators. CollAFL [12] introduces an instrumentation strategy to avoid collision in coverage feedback. AFLFast [6] and Entropic [5] improve seed selection to select seeds that are most helpful to discover new behaviors for mutation. For seed mutation, MOpt [23] collects information on mutation operators and schedules them. Many fuzzers [19, 27, 31, 33, 40] combine symbolic execution to help fuzzing to pass some complicated constraints. Many fuzzers [4, 7, 11, 21, 28, 39] utilize taint analysis to assist fuzzing, which will be introduced in the following texts. On the other hand, many works focus on improving fuzzing speed by improving instrumentation [24, 34, 35, 43]. Untracer [24] removes the instrumentation for tracing basic blocks that have been explored. Zeror [43] introduces a self-modifying tracing mechanism to provide zero-overhead instrumentation. Odin [35] provides on-demand instrumentation to dynamically adjust instrumentation on the fly.

DODRIO focuses on fuzzing parallelization. Its approach is orthogonal to the fuzzing strategies and can parallelize mutation-based fuzzers. It accelerates parallel fuzzing in covering basic blocks by reducing redundancy actions, rather than optimizing instrumentation to improve execution speed.

Taint Analysis Based Fuzzing. Taint analysis assists fuzzing by identifying critical bytes that influence the branching behavior, and fuzzers can selectively mutate them to improve efficiency. Taint analysis can be categorized into propagation-based and inference-based approaches. Propagation-based taint analysis [7, 28] assigns distinct labels to each input byte and propagates these labels using predefined propagation rules during execution. VUzzer [28] utilizes taint analysis to identify the bytes associated with magic value validations. Angora [7] tracks the bytes propagated into each path constraint and strategically mutates these bytes for solving constraints. Inference-based taint analysis [4, 11, 15, 21, 39] infers the critical bytes related to specific constraints by changing input bytes and monitoring program state changes. REDQUEEN [4] specifically targets the identification and resolution of magic values and checksums. GREYONE [11] infers taint by mutating seeds and monitoring changes in the values of variables associated with path constraints. PATA [21] employs a path-aware taint inference technique, which enables a more granular analysis of taint propagation for each access to the constraint variable.

Although these fuzzers showcase effective mutation strategies, their throughput is still limited by the absence of parallel support. As a complement, DODRIO focuses on parallelizing the taint analysis process. Through real-time state synchronization, all instances operate as a cohesive unit, ensuring seamless coordination and cooperation. The distribution of computational workload among multiple instances enables each instance to undertake distinct tasks, thereby preventing redundant behavior and promoting efficiency.

Parallel Fuzzing. Conventional parallel fuzzing like AFL [41] coordinates different fuzzing processes by synchronous input seeds on a regular basis. Based on that, many fuzzers try to optimize synchronization to reduce resource costs in synchronization by carefully designing synchronization mechanisms. YEPAFL's master node pulls the seed queue of each slave node and the slaves synchronize only from the master node. μFuzz [9] breakdowns fuzzing loop into concurrent microservices. It partitions the state into different services to avoid synchronization. UltraFuzz [45] uses a database to synchronize fuzzing information from distributed instances. Seed synchronization is also a way to utilize the capabilities of different fuzzers, which is called ensemble fuzzing. EnFuzz, Cupid, and CollabFuzz [8, 13, 25] ensemble diverse fuzzers to increase the ability to generate seeds. Moreover, many fuzzers attempt to partition fuzzing tasks and distribute them among instances [9, 16, 18, 26, 30, 38, 44]. PAFL [18] synchronizes guiding information and divides tasks based on the bitmap to avoid overlapping actions. AFLTeam [26] and AFL-EDGE [36] distribute mutually exclusive tasks to different instances by dividing the call or control flow graph with static analysis. In addition, some techniques speed parallel fuzzing by improving implementations at low level. For example, Xu [37] proposes several new primitives to speed up AFL in parallel mode.

DODRIO aims to maintain the parallel fuzzing performance as close as possible to that of the single-core fuzzing test with the same resources. It requires access to accurate global information in real time and proper task distribution. Most parallel fuzzers (e.g., PAFL and YEPAFL) and ensemble fuzzers (e.g., EnFuzz and Cupid) maintain coverage bitmap for each instance and they will have a delay in state synchronization. Differently, DODRIO maintains two global coverage bitmaps to ensure real-time synchronization. Unlike PAFL, AFLTeam, and AFL-EDGE, DODRIO does not partition parallel tasks based on bitmaps or control flow graphs. Instead, it acquires the tasks needed to be executed in the single-core algorithm based on the global state and directly assigns them to idle instances.

8 CONCLUSION

This paper presents DODRIO, a parallel fuzzing framework that aims to reduce the redundant behaviors in parallel fuzzing and improve performance. First, it uses real-time synchronization to update the global state. Second, it utilizes load-balanced task dispatching to allocate different tasks to different instances. DODRIO outperforms the classical parallel mode and state-of-the-art parallelization fuzzing work PAFL and YEPAFL on tested real-world programs in speeds in covering basic blocks, covered basic blocks, and bug triggering. In our future work, we plan to utilize DODRIO to parallelize symbolic execution based fuzzers to solve difficult constraints.

ACKNOWLEDGMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002) and China Postdoctoral Science Foundation (2023M731953).

REFERENCES

[1] 2020. FuzzBench: Fuzzer Benchmarking as a Service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html. [Online; accessed June 12, 2024].

[2] 2020. Fuzzer-test-suite: Set of tests for fuzzing engines. https://github.com/google/fuzzer-test-suite. [Online; accessed June 12, 2024].

[3] 2022. Scaling. https://hpc-wiki.info/hpc/Scaling. [Online; accessed June 12, 2024].

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[5] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1032–1043.

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725.

[8] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1967–1983.

[9] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. 2023. μFUZZ: Redesign of Parallel Fuzzing using Microservice Architecture. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security'23)*.

[10] Joe W Duran and Simeon Ntafos. 1981. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 179–183.

[11] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA. https://www. usenix. org/conference/usenixsecurity20/presentation/gan*.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[13] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference*. 360–372.

[14] Honggfuzz 2024. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: June 12, 2024.

[15] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 475–485.

[16] Yang Li, Han Zou, and Hongzhi Liu. 2020. A High Efficient Technology for Parallel Fuzzing. In *Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence*. 29–33.

[17] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-oriented DBMS fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 668–681.

[18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, USA, November 04-09, 2018*. 809–814.

[19] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2019. DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[20] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 562–566.

[21] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA*. 154–170.

[22] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[23] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[24] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[25] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*. 1–7.

[26] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. 2021. Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1337–1341.

[27] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[30] Congxi Song, Xu Zhou, Qidi Yin, Xinglu He, Hangwei Zhang, and Kai Lu. 2019. P-fuzz: a parallel grey-box fuzzing framework. *Applied Sciences* 9, 23 (2019), 5100.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[32] Ari Takanen, Jared D Demott, and Charles Miller. 2008. *Fuzzing for software security testing and quality assurance*. Artech House.

[33] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 61–64.

[34] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference*. 147–159.

[35] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1010–1024.

[36] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. 2021. Facilitating parallel fuzzing with mutually-exclusive task distribution. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17*. Springer, 185–206.

[37] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2313–2328.

[38] Jiaxi Ye, Bin Zhang, Ruilin Li, Chao Feng, and Chaojing Tang. 2019. Program state sensitive parallel fuzzing for real world software. *IEEE Access* 7 (2019), 42557–42564.

[39] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE.

[40] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[41] Michal Zalewski. 2015. American fuzzy lop.

[42] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. 2023. Daisy: Effective Fuzz Driver Synthesis with Object Usage Sequence Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 87–98.

[43] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 858–870.

[44] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, and Qidi Yin. 2020. Unifuzz: Optimizing distributed fuzzing via dynamic centralized task scheduling. *arXiv preprint arXiv:2009.06124* (2020).

[45] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, Qidi Yin, and Xu Han. 2022. UltraFuzz: Towards Resource-saving in Distributed Fuzzing. *IEEE Transactions on Software Engineering* (2022).

Received 2024-02-08; accepted 2024-04-18