μFUZZ: Redesign of Parallel Fuzzing using Microservice Architecture

Yongheng Chen[†], Rui Zhong[⋆], Yupeng Yang[†], Hong Hu[⋆], Dinghao Wu[⋆], Wenke Lee[†]
[†]*Georgia Institute of Technology* [⋆]*Pennsylvania State University*

Abstract

Fuzzing has been widely adopted as an effective testing technique for detecting software bugs. Researchers have explored many parallel fuzzing approaches to speed up bug detection. However, existing approaches are built on top of serial fuzzers and rely on periodic fuzzing state synchronization. Such a design has two limitations. First, the synchronous serial design of the fuzzer might waste CPU power due to blocking I/O operations. Second, state synchronization is either too late so that we fuzz with a suboptimal strategy or too frequent so that it causes enormous overhead.

In this paper, we redesign parallel fuzzing with microservice architecture and propose the prototype μFUZZ. To better utilize CPU power in the existence of I/O, μFUZZ breaks down the synchronous fuzzing loops into concurrent microservices, each with multiple workers. To avoid state synchronization, μFUZZ partitions the state into different services and their workers so that they can work independently but still achieve a great aggregated result. Our experiments show that μFUZZ outperforms the second-best existing fuzzers with 24% improvements in code coverage and 33% improvements in bug detection on average in 24 hours. Besides, μFUZZ finds 11 new bugs in well-tested real-world programs.

1 Introduction

In recent years, fuzzing has been widely adopted to detect security bugs [17, 36, 76, 79]. Compared with other program analysis techniques, fuzzing ensures higher throughput while requiring less manual effort and pre-knowledge of the target software. In addition, fuzzing is demonstrated to be practical for detecting security issues in complex, real-world programs [8, 79]. Thus, considerable computing resources are used for fuzzing in industry. For example, Google implemented clusterfuzz [4] in 2016, and over 36,000 bugs have been found through this project.

To improve the fuzzing efficiency, researchers propose various optimizations to enhance fuzzers' internal components [43, 50]. For instance, several projects implement grammar-based, adaptive or unified mutators to generate more valid, effective and diverse test cases [28, 46, 73, 83]. Hybrid fuzzing utilizes heavy program analysis techniques to extract useful information to help explore program state space [16, 59, 61, 67, 78]. Various algorithms are developed to adjust the input priority to make a balance between input space exploration and exploitation [19, 70, 80]. Researchers also design and implement different feedback mechanisms to promote the fuzzing speed and effectiveness [29, 51, 69]. These internal improvements have dramatically increased the performance of a single fuzzing instance.

In addition to improving internal procedures, researchers also set sights on parallel fuzzing. The goal of parallel fuzzing is to make full use of resources and detect more bugs within a shorter time than single-instance fuzzing. For example, as fuzzing shows its ability in bug detection, many companies such as Google and MicroSoft decide to invest numerous resources (*e.g.*, CPU and memory) in fuzzing [4, 5, 8, 10]. State-of-the-art parallel fuzzing approaches share a similar architecture [2, 58, 72, 79]. They launch multiple fuzzing instances in separate processes and periodically perform corpus synchronization. Each instance follows the original logic of the underlying *single-instance fuzzer*, which is designed to run as a single instance. For example, it adopts a serial fuzzing loop which first takes one test case from the input queue, then mutates the input to generate new ones, and finally runs the program with the new input while collecting feedback. Each instance maintains its own fuzzing states such as the code coverage bitmap and a corpus of interesting test cases. The advantage of this parallel-fuzzing architecture comes from state synchronization, which allows one instance to catch up on the latest progress from other instances. In this way, all instances contribute to the program state exploration and bug detection.

However, we identify two limitations in the current parallel fuzzing architecture. First, the existing architecture is built on top of single-instance fuzzers, whose fuzzing logic may not be suitable for parallel fuzzing purposes. These single-instance fuzzers adopt a serial synchronous loop, where the input generation and consumption must follow the order. Once a procedure (*e.g.*, test case execution) in this loop gets blocked (*e.g.*, by file or network I/O), the whole instance gets stuck. The CPU bound to the fuzzing instance will be idle or spinning (*e.g.*, keep looping for a lock access) until the blocking operation completes. As parallel fuzzing runs multiple instances at the same time and introduces more I/O by synchronization, the instances are more likely to get stuck, wasting more CPU cycles. We should reuse the wasted CPU cycles to fully utilize the computation power.

Second, existing approaches periodically synchronize the corpus from each other to allow instances with slow progress to catch up. The synchronization will update the local fuzzing states for all instances so that they can use the latest information to make globally beneficial fuzzing decisions. However, these state updates are not timely enough. In the time window between two consecutive synchronizations, each instance has to use the local information to make decisions. Since local information could be out-of-date, such decisions are not necessarily beneficial from the global perspective. After running fuzzing instances for a long time, the accumulated non-optimal decisions could significantly reduce the fuzzing efficacy. Increasing the frequency of synchronization could mitigate this problem. However, as demonstrated in the previous work [75], frequent synchronization brings heavy overhead, which will reduce the fuzzing efficiency.

To overcome the limitations caused by the current architecture, we need to redesign fuzzing tools to reduce the burdens of serialization and synchronization. Fortunately, we find our opportunity in *microservice architecture* [7]. Microservice architecture organizes tasks in a set of loosely coupled, self-contained services that can run concurrently with others. If no service is blocked, all services collaborate with each other according to the loose dependency. Once a running service is blocked, other services can take over the computing resources to make individual progress. Moreover, each service will maintain its own state and only needs to share minimal information with others in rare cases. Most of the time, each service can make globally optimal decisions.

In this paper, we propose μFUZZ, a parallel fuzzing framework using the microservice architecture. To adopt this new architecture, we break the current serial fuzzing loop into four microservices, *i.e.*, corpus management, test case generation, test case execution, and feedback collection. Each microservice is self-contained and can schedule parallel workers by itself. We further design an output cache mechanism to reduce the coupling between different services (*i.e.*, decouple input generation and consumption). In this case, if one consumer service gets stuck, the producer service can still make progress and save results into the cache. Similarly, the consumer services can retrieve results from the caches even if the producer service gets stuck. This effectively addresses the CPU cycle wasting issue since each microservice is loosely coupled and can replace the blocked service for execution.

To address the challenges caused by synchronization delay, we design two levels of state partition in μFUZZ. First, μFUZZ splits the global state into different service states so that each service can use its state locally. For example, the coverage bitmap will be put into the feedback collection service as it will evaluate the code coverage and update the bitmap according to the execution status. Second, different workers in each service handle unique parts of the service states. Accumulating all worker states will obtain the service states. These partitions avoid the state synchronization among the workers and enable each service worker to use up-to-date information to make overall good decisions.

We implement μFUZZ in 9534 lines of Rust code, consisting of the concurrent infrastructure (*i.e.*, the asynchronous runtime) and the fuzzer. The concurrent infrastructure is built on top of Tokio [3], a well-tested asynchronous runtime library. For the fuzzer, we adopt the fork-server execution, havoc mutation, and edge coverage feedback from `AFLplusplus`, and use a simple round-robin algorithm that favors test cases finding more new code for seed selection.

To understand the effectiveness of our new design, we evaluate μFUZZ on two popular benchmarks: Magma [38] and FuzzBench [48]. We compare μFUZZ with the state-of-the-art parallel fuzzers, including `AFLplusplus`, `AFLEdge` and `AFLTeam`, and find μFUZZ can explore 24% more program states and 33% more bugs than the second-best fuzzer in 24 hours. Besides, our experiments show that different aspects of the microservice architecture contribute to the improvement of μFUZZ. Moreover, μFUZZ found 11 new bugs in well-tested real-world programs.

In summary, this paper makes the following contributions:

- We propose a parallel fuzzing framework with microservice architecture that well utilizes CPU power even with blocking I/O and avoids state synchronization.
- We implement the prototype, μFUZZ, of our system to effectively perform parallel fuzzing.
- We compared μFUZZ with state-of-the-art fuzzers, and the results show that μFUZZ can find 24% more new coverage and 33% more bugs in 24 hours than the second-best fuzzer.

We release the code of μFUZZ in `https://github.com/OMH4ck/mufuzz`.

2 Problem

In this section, we first briefly describe how state-of-the-art parallel fuzzing approaches work and then discuss their limitations. Next, we show the potential of using microservice architecture to mitigate the limitations of parallel fuzzing. Finally, we present our novel approach to solving the problem.

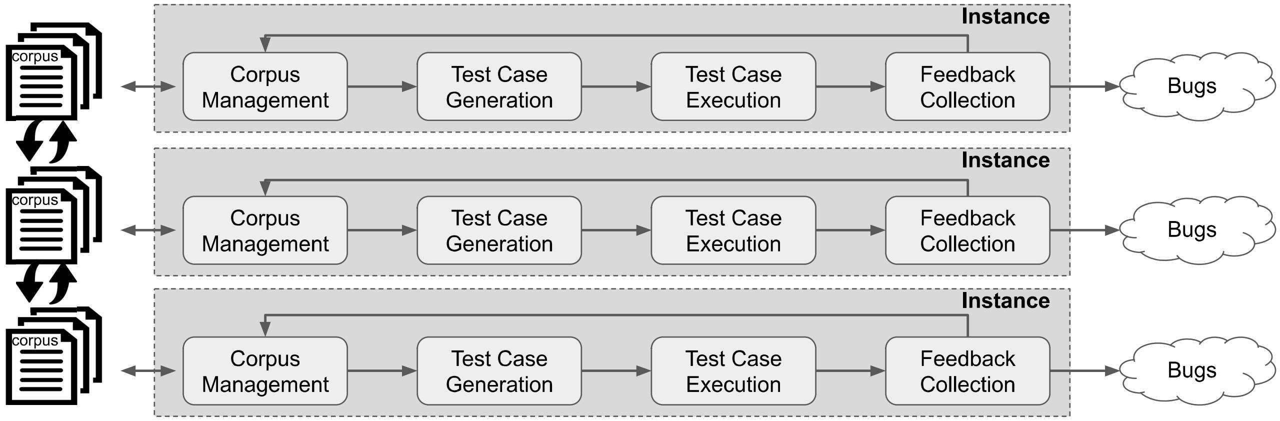


Fig. 1: The state-of-the-art parallel fuzzing approach. The fuzzer spawns multiple instances and runs them in parallel. Each instance is self-contained and functionality-complete. They maintain their own local fuzzing states, such as the corpus and coverage bitmap. Most of the time, the instances work independently as if there were no other instances. Occasionally, the instances perform corpus synchronization with each other to share their fuzzing progress.

2.1 How Existing Parallel Fuzzing Works

To better test complex programs with time constraints [5, 42], many fuzzers [2, 6, 22, 35, 79] support parallel fuzzing mode to boost the fuzzing performance. The state-of-the-art approach is to run multiple fuzzing instances of the same fuzzer independently on multiple CPU cores. The instances perform periodic corpus synchronization with each other because the corpus represents the fuzzing progress of an instance. Synchronizing the corpus allows the latest progress made by one instance to be caught up by the others and guide their work [79]. As shown in Fig. 1, each instance maintains a local seed corpus. Most of the time, these instances run independently, as if there are no other instances. Occasionally, they check others' seeds and copy those that trigger new code to their own corpus. Advanced parallel fuzzing approaches either run instances of different fuzzers to combine their capability [25, 37, 55] or further optimize the corpus distribution strategy by partitioning the synchronized corpus among instances to avoid duplicated fuzzing efforts [44, 58, 72].

Fuzzing State. Corpus synchronization improves fuzzing because the corpus is part of the fuzzing state. We define the fuzzing state of a fuzzing instance to be the minimum information to represent its full fuzzing progress. They might include the corpus, average running time of the test case executing, seeds of the random number generator, etc.

2.2 Limitation of Existing Approaches

Existing parallel fuzzers maintain a local fuzzing state in each instance and perform state synchronization periodically. Such approaches mainly have two problems: First, it aggravates the problem of wasting CPU cycles due to the serial design of the underlying fuzzer. Second, the global fuzzing state cannot be synchronized to each instance timely or efficiently, resulting in suboptimal performance.

Table 1: The percentage of wasted CPU cycles by blocking I/O in single instance fuzzing.

Target	`lua`	`PHP`	tcpdump	MySQL
Wasted Cycles (%)	1.2%	0.5%	28.7%	60.3%

Table 2: The percentage of wasted CPU cycles by blocking I/O in parallel fuzzing. We measure the blocking cycles in the main fuzzing instance with `AFLplusplus` during corpus synchronization with 1-second, 1-minute and 30-minute synchronization intervals and different number of instances.

Target	`lua`			`PHP`		
Interval\Instance	20	40	60	20	40	60
1 sec	3.18%	7.29%	8.74%	13.54%	25.79%	29.70%
1 min	0.13%	0.30%	0.31%	1.34%	2.47%	3.34%
30 min	0.07%	0.08%	0.15%	0.58%	0.76%	1.01%

CPU Cycles Wasted due to Blocking I/O. Existing fuzzers run their instances in a serial synchronous loop [2, 35, 79]. For example, the fuzzing pipeline of `AFLplusplus` is as follows: Select a test case, mutate it, execute it, check the execution feedback, and loop. If any of the steps are blocked by I/O, the other steps can do nothing but wait. Therefore, such a design might suffer from performance degradation in the existence of blocking I/O. I/O can come from two sources. First, the tested program can involve heavy blocking I/O (e.g., a compression application might do heavy file I/O.). During the execution phase, the fuzzing loop can get stuck, waiting for the I/O to complete. Since the fuzzing loop is synchronous, the CPU cannot perform other fuzzing tasks but wait, wasting CPU cycles. Second, when fuzzing with multiple instances, state synchronization might also bring in lots of I/O [12]. Take `AFLplusplus` as an example. When running in parallel mode, each instance periodically checks and synchronizes the corpus with other instances in a shared folder. This has been shown to bring lots of I/O, such as shared folder locking and file copying, which hurts the fuzzing performance [75].

We performed quick experiments on four popular programs to investigate these two sources of I/O. We measure the percentage of CPU cycles wasted by blocking I/O, where the CPU is either idle or spinning waiting for gaining locks or I/O completion. Table 1 shows the result in single instance fuzzing. As we can see, if the tested target involves lots of blocking I/O (*e.g.*, `tcpdump` calls the system call `poll` and waits), the wasted percentage can be significant. For targets without much I/O (*e.g.*, `lua` and `PHP`), we futher evaluated them for the main fuzzing instance (*i.e.*, the instance that performs synchronization with all other instances) in parallel fuzzing with respect to different synchronization intervals and number of instances. As shown in Table 2, even for targets with little inherent I/O operations, if we perform heavy state synchronization (with more cores or higher frequency), the introduced I/O significantly degrades the CPU utilization. Simply spawning more instances on the same CPU (*i.e.*, oversubscription) cannot solve the problem because it might introduce more resource contention such as context-switching and again hurt the performance.

Fuzzing State Not Timely Synchronized. The instances maintain their local fuzzing states and perform periodic synchronization. Before the next synchronization, they use the possibly outdated states and fuzz with strategies which are locally optimal but can be globally suboptimal. On the other hand, it is not feasible to perform synchronizations too frequently as it incurs a significant overhead [75].

We did a quick experiment to verify our hypothesis. We used `AFLplusplus` to fuzz `QuickJS` [9], a popular JavaScript engine. As a comparison, we fuzzed with one instance of `AFLplusplus` for ten hours and ten instances in parallel for one hour, respectively. The measured metrics include the code coverage and the number of interesting test cases that are further selected for fuzzing. The result shows that if we fuzz with a single instance for 10 hours, about 13,700 new program paths are found, and about 80% of the interesting test cases are further used for fuzzing. However, when fuzzing with ten instances in parallel for an hour, we only find about 6,700 new program paths, which is only 49% of that of a single instance. About only 40% of the test cases are further selected for fuzzing. We assume the fuzzing strategy of the single-instance fuzzer is optimal. That means the ten instances use suboptimal strategies and duplicate their works on similar test cases, while the globally optimal strategy is to explore test cases diversely. Similar results are also found in [72].

To further verify that the performance gap is caused by synchronization delay, we change the synchronization frequency of `AFLplusplus` and measure the change in the fuzzing performance in terms of code coverage. More specifically, we fuzz `QuickJS` with ten `AFLplusplus` instances for one hour by setting their synchronization frequency per hour from 2 (`AFLplusplus`'s default setting) to 40,000 (which performs synchronization after every test case execution). The result is shown in Fig.2. As we can see, if the frequency is too low, the code coverage is also low because the instances are using sub-

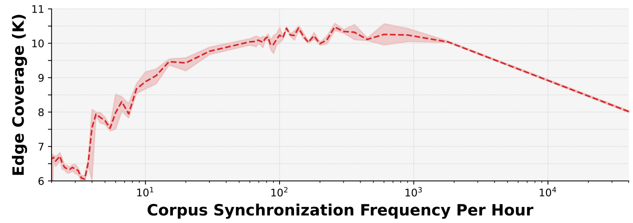


Fig.2: Code coverage of 10 `AFLplusplus` instances testing `QuickJS` with different synchronization frequency in an hour.

optimal fuzzing strategies. If the synchronization frequency is high, the code coverage also drops dramatically because the overhead of synchronization is too high. However, even the best result in the curve is still much worse than that of the single instance fuzzing. This means that simply changing the synchronization frequency does not solve the problem.

From the above discussion, we want a parallel fuzzing framework that supports concurrency to better utilize CPU cycles even in the existence of I/O and can synchronize instances' states timely with little overhead so that we fuzz with up-to-date states and make good decisions. However, it is difficult to do so on top of existing fuzzers with monolithic serial architecture. We need a different architecture.

2.3 Microservice Architecture

We find microservice architecture [7] fits parallel fuzzing well and can potentially mitigate its current limitations. First, microservice architecture structures the application as a set of small, loosely coupled, collaborating services. These services run concurrently with others. For parallel fuzzing, we can break the different phases in the serial fuzzing loop into concurrent services, where we might run other services if one gets stuck. Second, the services are self-contained (*i.e.*, it does not rely on others to finish its job), which means it does not need to synchronize with others. For parallel fuzzing, the services can be self-contained if each of them focuses on a single functionality of fuzzing (*e.g.*, test case generation). And we do not need state synchronization among the services. Third, inside a service, we can easily scale the capability by creating multiple instances and partitioning the service data among the instances. For parallel fuzzing, we can create multiple workers inside a service to achieve parallelism and partition the service state maintains among the workers. And these workers do not need to synchronize with each other because their states have no overlap. We only need to ensure that the workers can work independently using their local states and still achieve a good aggregated result.

2.4 Our Approach

This paper aims to design a parallel fuzzing framework that embraces concurrency to better utilize CPU power even with blocking I/O and avoids synchronization but still gets overall

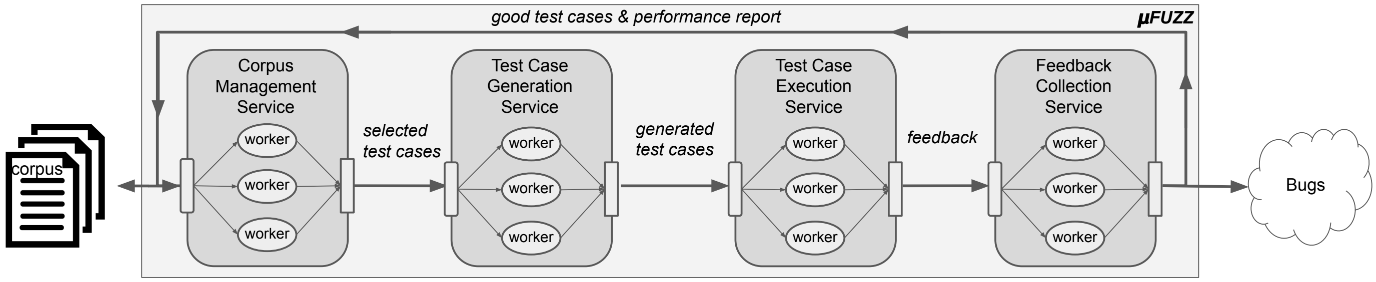


Fig. 3: Overview of μFUZZ. Instead of running multiple fuzzing instances and performing periodic synchronization, μFUZZ breaks the traditional monolithic architecture into microservices. The new architecture consists of four self-contained services, each maintaining a partition of the fuzzing state. The services are loosely dependent on each other using output caching. Inside each service, we run multiple workers to exploit parallelism.

great performance. We achieve our goal in two steps: redesigning the fuzzing framework with microservice architecture and partitioning the fuzzing state. Microservice architecture adds concurrency to the framework and enables the fuzzer to effectively utilize the CPU power in the existence of I/O. Fuzzing state partition allows the instances to fuzz with locally optimal strategy and still achieve an overall globally great performance without synchronization.

Redesign with Microservice Architecture. We break the traditional serial fuzzing loop into four services based on functionality: Corpus management, test case generation, test case execution, feedback collection. The whole fuzzing state is also partitioned among the services in a way that each service only needs its partition to function and is thus self-contained. However, these services are still tightly coupled: every service produces output for other services to consume and vice versa. Instead of running the services synchronously, we utilize output caching to decouple production and consumption so that they can run concurrently. When one service gets stuck, other services can still make progress and cache the outputs. After the stuck service is ready to run again, it can directly consume the cached outputs without waiting for the producer to generate them. In this way, we can better utilize the CPU power even with blocking I/O.

Partition the Fuzzing State. We have performed the first level of fuzzing state partition by breaking down the monolithic structure. Now each service maintains its own service state. However, if the state is shared by the workers, we still need state synchronization among the workers. We further partition the service state among the workers to avoid synchronization. We use two rules to guide the partition. First, each partition of the state should be functionality-complete, which means a worker can finish its job without using others' states. Second, if each worker adopts its locally optimal strategy, we expect to get a globally great (*i.e.*, close to optimal) aggregated result. In this way, the workers can run independently and do not need synchronization with others. Since we only have one global and distributive fuzzing state, the state changes are directly applied to the states inside the workers. Therefore, the workers always fuzz with the update-to-date global state and make good fuzzing decisions. This avoids the problem of periodic synchronization, which suffers from either high overhead or large synchronization lagging.

3 Design

Fig. 3 shows the overview of μFUZZ. We first break the traditional serial fuzzing loop into four services (§3.1). This step partitions the responsibility and the state of the fuzzer among different services so that they do not need state synchronization with each other. These self-contained services are the candidates for concurrency. Next, we utilize output caching to allow the services to run concurrently (§3.2) and achieve maximum parallelism with load balancing (§3.3). Then we further perform state partitioning among workers so that each worker maintains a self-contained partition of the fuzzing state (§3.4). This allows the workers to avoid synchronization with each other but still get an overall great aggregated result. Finally, we connect the services together with zero-copy communication (§3.5) to achieve efficient parallel fuzzing.

3.1 From Monolith to Microservice

As the first step to support concurrency, we break the monolithic serial fuzzing loop into multiple services, whose structures are shown in Fig. 4. We use the following guidelines from the microservice architecture to conduct the breakdown. First, each service should focus only on one core functionality of the fuzzing (*i.e.*, be micro). Second, the services should be self-contained, which means they should not rely on the states of other services to function. If any part of the fuzzing state is used by a service, then it should be maintained by the service. As a result, we classify four core functionalities from the fuzzing loop and break them into four services, which are listed below:

Corpus Management Service. It is responsible for performing test case scheduling and maintaining a corpus of interesting test cases and their associated metadata (*e.g.*, performance scores). The corpus can include test cases finding new code coverage or triggering new bugs, etc. Based on the metadata

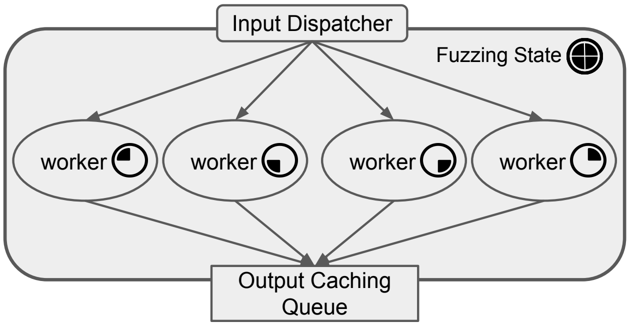


Fig. 4: The internal structure of a service in μFUZZ. Each service has a fuzzing state, an input queue dispatcher, an output caching queue, and some workers. The fuzzing state is partitioned among the workers. The input queue dispatcher accepts requests from other services and dispatches them to the workers. The workers handle the inputs in parallel and send the results to the output caching queue. The consumer services consume these results when they are ready.

of the test cases, the scheduling algorithm prioritizes those can better explore the program for test case generation.

Test Case Generation Service. It generates new test cases to fuzz the tested program either from scratch or by mutating existing ones. For example, it can utilize the BNF grammar to generate structured inputs or bit-flip existing test cases to generate new variants.

Execution Service. It executes the tested target with the generated test cases and generates necessary feedback such as the code coverage information and whether the tested program crashes or timeouts during the execution.

Feedback Collection Service. It collects the feedback from the execution service and classifies whether the feedback is interesting or not. This information can be used to decide whether a test case should be added to the corpus. It can also generate fuzzing statistics in different metrics for other services to improve their strategies. For example, it can calculate how many interesting test cases are generated from a specific seed and report that to the corpus management service. The corpus management service can then utilize the statistics to update the performance scores of the corresponding test cases and fine-tune its scheduling algorithm.

We see these services are dependent on each other and form a loop: each service consumes some outputs from other services and also produces some for them. Without further changes, these services still need to run in a serial way that one getting stuck blocks the overall progress. We need to loosen the coupling between the services so that they can run concurrently and mitigate CPU cycle wasting, as described in the next section.

3.2 Concurrency by Output Caching

Each service is both a producer (produces inputs for other services) and a consumer (consumes outputs from other services). We decouple the production and consumption of each service by output caching so that the services can run concurrently. More specifically, we connect the services with an output caching queue, as shown in Fig. 4. When a service produces some results, it first sends them to the output queue instead of to the consumer service directly. If the consumer service is busy temporarily, the results just stay in the queue, and the producer service is free to produce more results. Once the consumer service is ready to process new inputs, it can directly fetch the cached ones from the output queue. In this way, services can run concurrently. When one gets blocked, others can still run and make progress.

Congestion Control. If we allow unlimited output caching, one potential problem is that one service might keep generating outputs and fully occupy all the CPU cores. Under this situation, other services have no chance to run and consume these cached outputs. And the fuzzing cannot make overall progress. For example, the corpus management service can keep selecting test cases for mutation and send them to the queue. And the test case generation service can not consume them as all the CPU cores are busy running the corpus management service. Therefore, we adopt congestion control by limiting the maximum number of cached results in the queue. When the producer service finds that the output queue is full, it knows that the consumer service needs more time to process the cached outputs. Then it will yield to the scheduler so that other services can run. In this way, the rate of production and consumption can reach a dynamic balance, and the fuzzing can make smooth progress continuously.

3.3 Parallelism by Load Balancing

To fully utilize the computation power of multiple cores, each service of μFUZZ can run multiple workers in parallel. To achieve maximum parallelism, we set the number of workers to be the number of cores, and we perform load balancing with an input dispatcher to keep all workers busy.

The input dispatcher maintains a first-in-first-out queue of idle workers and adopts two strategies of load balancing: "first come, first served" and dynamic input resizing. We define a worker as idle if it is ready to process but not currently processing inputs. Such workers notify the input dispatcher, which puts them into the back of the queue in order. Whenever an input arrives, the input dispatcher tries to pop an idle worker from the front of the queue and dispatch the input to it, which is "first come, first served." If the queue is empty, which means all workers are busy, the input dispatcher will wait for a worker to become idle. This strategy works well for most cases. However, the sizes of the incoming inputs are not fixed, and sometimes they can be very large. If we simply dispatch an input to one worker, it might result in one worker processing a large input while other workers are idle. To avoid this situation, we further perform dynamic input resizing before dispatching. If the size of the arrived input is larger than a threshold value and there are more than one idle

Algorithm 1: State Partition & Update

```
Procedure StatePartition(state)
    numWorkers ← The number of workers in the service;
    if state is fixed-sized then                    // Static paritition
        partitions ← Evenly partition the state based on numWorkers ;
    else                                            // Dynamic partition
        state ← RandomlyShuffle(state) ;
        partitions ← Evenly partition the state based on numWorkers ;
    Distribute one partition to one worker ;
    The worker tags the partitioned state with its ID and store it
Procedure StateUpdate(stateUpdateRequest)
    id2WorkerMap ← Map(ID, worker);
    id ← ExtractID(stateUpdateRequest) ;
    worker ← id2WorkerMap.Get(id) ;
    worker.ProcessRequest(stateUpdateRequest)
```

worker, we partition the input evenly based on the number of idle workers and dispatch one partition to one idle worker. With the two strategies, we can achieve maximum parallelism by keeping the workers' workload balanced dynamically.

3.4 Avoid Synchronization by State Partition

As discussed, if the fuzzing instances maintain their local states and rely on periodic synchronization, they suffer from either synchronization lagging or high overhead. Therefore, we adopt state partition to maintain only one global state without synchronization.

μFUZZ performs two level of state partition: In the first level μFUZZ partitions the fuzzing states among its services so that the services can work independently. In the second level μFUZZ further partitions each service state among the workers so that each worker can work independently. We already perform the first level by breaking the fuzzing loop into services. μFUZZ partitions its two fuzzing states in two of its services: The corpus management service maintains the interesting testcases with their metadata (e.g., the execution time) and the feedback collection service maintains the code coverage bitmap. The test case generation service and the execution service are stateless. We further perform the second level state partition inside each service. The goal is that each worker maintains a unique partition of the service state and can run independently without synchronizating with others. Meanwhile, accumulating the results from all workers still gives a good overall result.

State Partition. We perform two types of partition based on the size of the fuzzing states: static partition for fixed-sized states and dynamic partition for variable-sized states, as described in algorithm 1. For fixed-sized states, we partition them evenly based on the number of workers. Each worker gets a partition of the same size to achieve workload balance. Since we know the size of the state, we can do it statically. For example, suppose that we have a 1000-byte bitmap and 10 workers in the feedback collection service. By static partition, each worker should maintain 100 bytes (e.g., the first worker maintains byte 0 to byte 99, and the second maintains byte 100 to 199). For variable-sized states, we partition them dynamically whenever new states are found, and distribute them evenly and uniformly at random to the workers. The intuition is that each partition maintained by the workers follows the same or similar data distribution as the whole service state. In this way, each worker can work independently but still get a good accumulated overall result with high probability. For example, if we find 100 new interesting test cases and distribute them randomly to 10 workers. Suppose that the global optimal strategy is to pick 10 test cases with the best performance scores for fuzzing, such strategy can be approximated by asking each worker to pick its best test case and combining them. μFUZZ performs static partition on the coverage bitmap in the feedback collection service and dynamic partition on the corpus in the corpus management service.

Result Accumulation. We need to accumulate outputs for some services to generate an overall result. For the feedback collection service, we simply aggregate the count for the "insteretingness" for the test cases: All the workers will check its partition of bitmap and output whether a test case triggers new bits. If all the workers say no, then the test case will be discarded. Otherwise, it will be sent to the corpus management service and added to the corpus. For the corpus management service, we forward the outputs without any changes since they already approximate a good result by simple aggregation as discussed above. In μFUZZ, the output caching queue performs result accumulation since all outputs go through it.

State Updates. In μFUZZ, the fuzzing state is updated by its maintaining serivce worker, and the state update requests come from other services. For example, a worker in the corpus management service will update a test case's performance score upon receiving feedbacks from the feedback collection service. However, the service state has been partitioned across the workers. When a service receives state update requests, we need to tell the input dispatcher which worker should process them. For statically partitioned states like the coverage bitmap, we use the partition boundary as the workers' unique identifier (ID). We then check the range of the bit offsets and figure out the corresponding worker. For dynamically partitioned states, we assign each worker with a unique number as the identifier, and all the states within the worker are tagged with the ID. Such IDs will be carried along the fuzzing loop and guide the input dispatcher. For example, every test case in the corpus management service will be tagged with its maintaining worker's ID. Therefore, the test cases sent to the mutation service will carry the ID and pass through to the feedback collection service. In this way, when the feedback collection service sends the performance score update requests to the corpus management service, the input dispatcher can check the IDs and dispatch the requests to the maintaining worker. The overall workflow in shown in algorithm 1.

3.5 Zero-Copy Communication

As mentioned before, we break the fuzzing loop into different services, and each service consumes the outputs from other services and produces some for them. Considering the fast speed of fuzzing, the amount of passing data can be huge and thus potentially introduce high communication overhead. Therefore, we design a safe zero-copy mechanism to reduce communication overhead. Specifically, we utilize pointer passing with shared memory to pass only a constant size of data (*i.e.*, a pointer and a data size) regardless of the amount of generated outputs and unique ownership to enable safe access to data across services. For example, suppose the average size of the generated test cases is 1,000 bytes long, and the test case generation service generates 1,000 new test cases per second. Assuming we always copy the data from one service to another, the required data copying from the test case generation service to the execution service will be 1,000,000 bytes per second. The number will keep going up if we fuzz with more cores. However, if we can pass a pointer to the data, we pass only eight bytes on a 64-bit system.

Pointer Passing with Shared Memory. To avoid unnecessary memory allocation and data copying between the producer and consumer services, we create shared memory between the services and pass the pointers to the shared memory instead. After the shared memory is set up (*e.g.*, using `mmap`), the producer service writes its outputs directly to the shared memory. To "pass" the data to the consumer, the producer simply passes a pointer to the data and the size of the data to the output queue. Afterward, the consumer can fetch the pointer and the size to perform accurate data access. In this way, regardless of the output size, we only need to pass the small constant-size pointers and integers.

Unique Ownership for Safe Access. Using shared memory poses a safety risk. Since the shared memory is accessible from multiple services, if we allow the services to access the memory at the same time, race conditions could happen. To address this, we wrap the pointers in unique ownership to ensure safe memory access. This unique ownership ensures that only one service can access the shared memory at any given moment. This makes sense because the consumer should only access the output after the producer finishes generating it, and the producer does not need to access it afterward.

4 Implementation

We implement μFUZZ in 9534 lines of code. Table 5 shows the breakdown.

Concurrent Runtime. We use `Tokio` [3] as the concurrent runtime of μFUZZ. The runtime is responsible for efficient task scheduling. Each worker in the services of μFUZZ is run as a task in the runtime. The number of workers per service is empirically set to be the number of cores to achieve maximum parallelism. Users can adjust the number accroding to their use cases (*e.g.*, perform a short time dry run to try different values and pick the best one). We maintain a queue of unfinished tasks to execute. If the runtime is looking for a task to run, it pops one from the front of the queue. When a service receives inputs, its workers will get notified, and μFUZZ will try to put the workers in front of the queue, which allows them to be picked up for execution sooner. After a worker finishes its work, we put it at the back of the queue so that workers from other services have a chance to run. If all the inputs are processed or the service gets stuck, μFUZZ will move to the next service with inputs to be processed.

Corpus Management. The corpus management service maintains a corpus of test cases and their performance scores used in the test case selection algorithm. The performance score of a test case reflects how many interesting variants it has generated. When a test case is added to the corpus, we assign it an initial score and adjust it according to the feedback. For example, if a mutated variant of a test case triggers a new code path, the score of the test case is increased. For test case selection, we sort the test cases by scores and select them in descending order with random skipping.

Test Case Generation. μFUZZ uses `AFLplusplus`'s havoc mutation as its test case generation, which performs unstructured bit flip and byte modification on existing test cases. Since test case generation and execution are in separate services, sending the mutated test cases one by one to the execution service will result in too much service switching. Instead, we send the new variants in bulk to the execution service to reduce the overhead.

Execution. The execution service adopts the popular forkserver executor in its worker [2, 79]. Each worker maintains a fork-server. When a worker receives an input to execute, it feeds the input into the fork server and requests a process fork. The forked process executes the target binary with the test case as input and generates the code coverage and execution status (*e.g.*, crash, timeout).

Zero-Copy Communication. We run all services of μFUZZ in the same process to share the memory address space. To support multiprocess run, we can use `mmap` to create shared memory. In this way, zero-copy communication can be achieved by simple pointer passing. We use Rust's `std::sync::Arc`, a thread-safe reference-counting pointer, to wrap our data. We achieve unique ownership by ensuring that the reference counter of the pointer is always one so that there is at most only one owner for the underlying memory at any moment.

5 Evaluation

Our evaluation aims to answer the following questions.

- Can μFUZZ outperform state-of-the-art parallel fuzzers? (§5.2)

- What is the contributions of μFUZZ's components in the fuzzing performance improvement? (§5.3)
- Can μFUZZ find new bugs in real-world programs? (§5.4)

5.1 Evaluation Setup

Benchmark. We use the state-of-the-art benchmark `Magma` [38] to evaluate μFUZZ in bug detection capability and code coverage. We use the corpus from `Magma` for all the targets and run them through `AFLplusplus`'s test case minimizers to remove redundant ones beforehand. We compare μFUZZ with three state-of-the-art fuzzers: `AFLplusplus` [2], `AFLTeam` [58], and `AFLEdge` [72]. `AFLplusplus` is the most popular fork of `AFL` with various improvements and is actively maintained. `AFLTeam` and `AFLEdge` are the most recent and the open-source advanced parallel fuzzers, which focus on partitioning fuzzing tasks to different instances and are thus good comparison for μFUZZ's state partition. `AFLEdge` and `AFLTeam` work by integrating with existing single-instance fuzzers. Therefore, we run `AFLTeam` and `AFLEdge` on top of `AFLplusplus` for a fair comparison. For new bug detection, we evaluate μFUZZ with programs from `FuzzBench` [48].

Environment Setup. We perform our evaluation on five machines, each with an Ubuntu 18.04 operating system, an Intel Xeon CPU E5-2680 v3 processor (48 virtual cores) and 256 GB RAM. We instrument the tested programs to test edge coverage. For the code coverage and bug detection experiments, we run the fuzzers with 40 fuzzing instances on 40 cores for 24 hours. For μFUZZ, we run 40 workers for each of the four services but still use only totally 40 cores, the same number as the other fuzzers. We apply `Magma`'s survival analysis to convert the recorded bug triggering time to bug survival time, which is the expected time a bug remains undiscovered [38]. A smaller survival time indicates a fuzzer can find the bug in shorter time. We run each set of experiment in a new docker container to reduce environment interference, repeat the process five times and report the average results to reduce the random noise.

5.2 Comparison against existing fuzzers

We compare μFUZZ against three state-of-the-art fuzzers, including the *de facto* `AFLplusplus` and the two most recent parallel fuzzers, `AFLEdge` and `AFLTeam`. To understand whether increasing the synchronization frequency of existing fuzzers can be a solution, we compare with AFLpp-FS, which is `AFLplusplus` performing synchronization every 30 seconds instead of 30 minutes. We use 30 seconds because it works as well as shorter intervals in code coverage, but with less overhead according to our experiments. The evaluated metrics include bug detection capability (the number of triggered bugs and their survival time) and edge coverage.

Bug Detection. As shown in Table 3, μFUZZ finds 20 bugs in 24 hours, while `AFLplusplus`, `AFLEdge`, `AFLTeam`, and AFLpp-FS find only 14, 15, 12, and 15 bugs, respectively. All the 17 bugs found by other fuzzers are also covered by μFUZZ, and μFUZZ found 12 of them using the shortest time. Three of the bugs (`PDF021`, `XML002`, `XML003`) are only found by μFUZZ. Additionally, AFLpp-FS finds all the 14 bugs found by `AFLplusplus` and one more, but it still finds five less than μFUZZ. This shows that more frequent synchronization can help improve the bug detection capability of `AFLplusplus` because the fuzzing instances can catch up with the latest progress earlier, but the improvement is limited.

Code Coverage. As shown in Fig. 5, on average μFUZZ identifies 24%, 41%, 80%, and 31% more new edges than `AFLplusplus`, `AFLEdge`, `AFLTeam`, and AFLpp-FS respectively. If the programs have a larger program state space to explore, μFUZZ can achieve higher code coverage improvement (*e.g.*, 40% more in `Poppler` and 37% more in `PHP` than `AFLplusplus`). Otherwise, the improvement of μFUZZ is smaller (8% more in `sndfile` than `AFLplusplus` and 2% less than AFLpp-FS). μFUZZ uses almost the same fuzzing strategies as `AFLplusplus` but has a higher code coverage. This is because `AFLplusplus` does not perform state partition and relies on corpus synchronization at long intervals (*i.e.*, 30 min). Between two synchronization, the fuzzing instances are not aware of the progress made by others. Interestingly, the coverage of AFLpp-FS is higher than `AFLplusplus` at the beginning (*i.e.*, in the first few hours) but lower in the end. We investigate the results and find the following reasons. When the fuzzing starts, the corpus is small and the program space is not well explored. An instance quickly finds a bunch of interesting test cases, but cannot explore all of them timely. Under this situation, more frequent synchronization allows other instances to catch up with the progress and help explore the interesting test cases. And a small corpus can be synchronized with low overhead. However, as the fuzzing goes, new code becomes harder to trigger and there are not as frequent progress updates as in the beginning, but AFLpp-FS still synchronizes the corpus frequently. Since the corpus has grown bigger, synchronization becomes more expensive, resulting in a slower increase in code coverage.

On the other hand, `AFLEdge` and `AFLTeam` perform state partition but still have worse performance than μFUZZ. We investigate their algorithms and execution status and find that they both follow a period "gather and partition" approach: Once per hour, they aggregate the corpus of all their instances and then perform partitioning based on heavy analysis. However, since we are running the experiment with 40 instances and the size of the aggregated corpus is large, the analysis takes a long time to finish. For example, we find that it takes `AFLEdge` more than three hours to finish one round of partitioning on `PHP`. By the time it finishes, the fuzzing has made three more hours' progress. The partition results might be obsolete and not necessarily beneficial to the latest fuzzing state. In com-

Table 3: Bug Detection Results in 24 Hours. We measure the bug detection capability in the number of identified bugs and their average *survival time*, which indicates the time a fuzzer needs to trigger the bugs. If the fuzzer cannot find the bug in 24 hours, we mark the survival time as ∞. The bug IDs are the unique identifiers for the inserted bugs in `Magma`. The time highlighted in green means the corresponding fuzzer is the fastest to find the corresponding bug. We sum up the number of the bugs found during any of the five runs in "Total Bugs Found". We exclude results for `openssl` and `PHP` because no fuzzers find any bugs in these targets in 24 hours.

Targets	Bug ID	μFUZZ	AFLplusplus	AFLEdge	AFLTeam	AFLpp-FS	μFUZZ-S	μFUZZ-P	μFUZZ-C
Poppler	PDF010	12h56m	19h18m	22h13m	05h31m	13h13m	∞	15h02m	17h40m
	PDF016	01m40s	16m40s	01m40s	16m40s	01m40s	03h21m	01m40s	09m10s
	PDF021	05h22m	∞	∞	∞	∞	∞	∞	13h51m
sndfile	SND017	01h24m	03h34m	02h51m	02h46m	16m45s	02h46m	03h54m	03h40m
	SND020	01h24m	04h15m	02h51m	02h46m	02h50m	02h46m	04h10m	02h54m
libxml2	XML002	12h57m	∞	∞	∞	∞	∞	14h31m	15h58m
	XML003	02h46m	∞	∞	∞	∞	∞	13h53m	06h10m
	XML009	16m40s	02h46m	02h46m	02h46m	02h46m	03h06m	02h46m	23m53s
	XML012	15h13m	∞	∞	∞	20h45m	20h07m	17h02m	19h49m
	XML017	01m40s	16m40s	01m40s	01m40s	01m40s	02h48m	06m40s	01h53m
SQLite	SQL002	02h46m	10h34m	07h28m	∞	03h16m	06h19m	17h23m	03h12m
	SQL018	02h46m	02h52m	05h14m	∞	03h40m	03h33m	05h16m	05h27m
lua	LUA004	02h47m	09h47m	19h11m	06h20m	16h40m	19h48m	02h48m	02h55m
libpng	PNG001	19h37m	∞	∞	∞	∞	∞	∞	∞
	PNG003	01m40s	03m25s	01m40s	02m20s	01m55s	17m20s	17m30s	01m40s
	PNG007	02h28m	02h50m	21h32m	23h05m	18h36m	06h03m	02h46m	03h03m
libtiff	TIF002	05h56m	∞	18h17m	∞	∞	∞	12h32m	∞
	TIF007	01m40s	16m40s	05h55m	01m40s	02h47m	05h34m	02h46m	02h51m
	TIF012	02h35m	09h26m	06h23m	02h46m	10h35m	14h29m	06h39m	02h55m
	TIF014	05h47m	03h11m	06h06m	04h22m	02h50m	12h06m	04h44m	03h01m
Total Bugs Found		**20**	**14**	**15**	**12**	**15**	**14**	**18**	**18**

parison, μFUZZ updates its state continously and timely, and thus avoids using obsolete states. Although μFUZZ's partition is not as comphrehensive as that of `AFLTeam` and `AFLEdge`, the timely state updates allow μFUZZ to make fuzzing decisions that fit the current state.

Overall, μFUZZ outperforms the four compared fuzzers in both bug detection and code coverage under parallel fuzzing. The fuzzing effectiveness of μFUZZ comes from both its architecture and state partition.

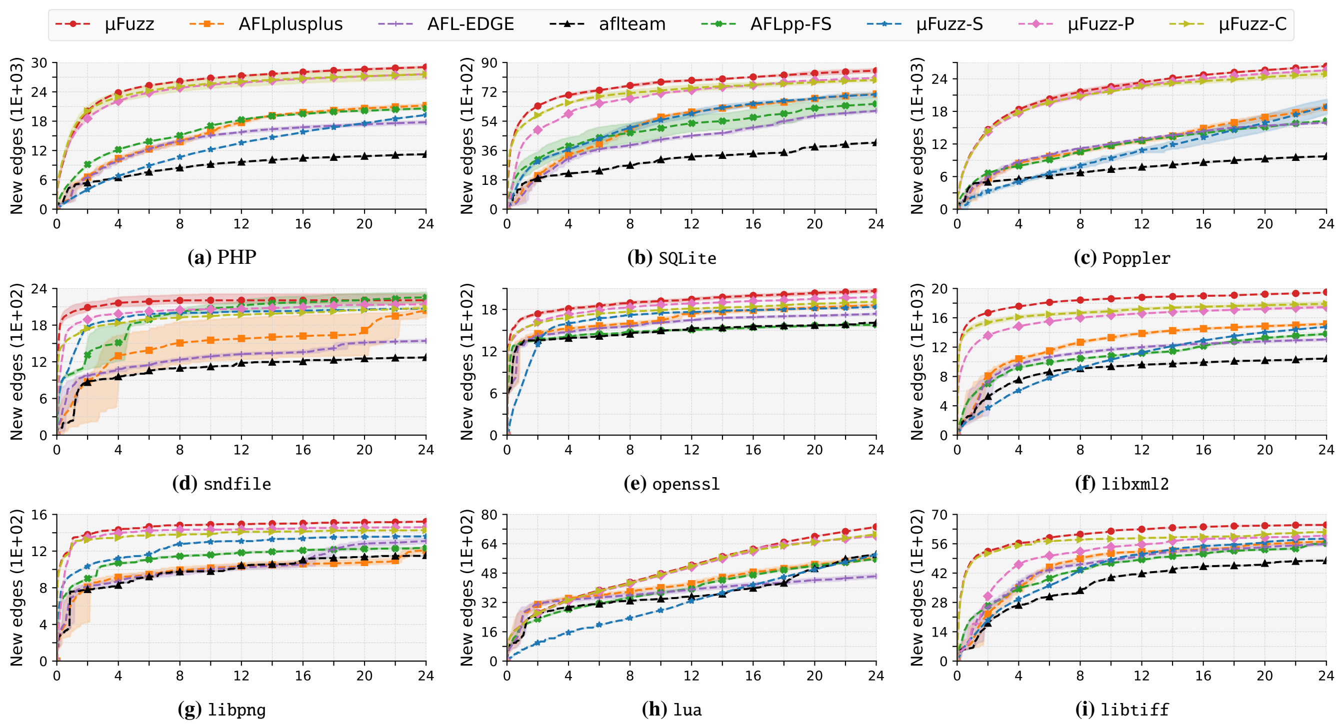
5.3 Contribution of Components

To better understand the contributions of μFUZZ's components, we compare μFUZZ with μFUZZ-S (μFUZZ without state partition), μFUZZ-C (μFUZZ without concurrency), and μFUZZ-P (μFUZZ without zero copy communication). More specifically, every worker of the corpus management service and the feedback collection service in μFUZZ-S maintains a copy of the global service state. Whenever there are state updates, μFUZZ-S will propagate the updates to all its workers. μFUZZ-C handles all the requests and generates output synchronously. μFUZZ-P passes full copies of data for all service communication instead of pointers.

Bug Detection. As shown in Table 3, μFUZZ successfully identifies 20 bugs in the targets, while μFUZZ-S, μFUZZ-C and μFUZZ-P finds only 14, 18 and 18 respectively. μFUZZ finds all the bugs that are found by the other three fuzzers with shorter time, and both μFUZZ-C and μFUZZ-P cover the bugs found by μFUZZ-S. Therefore, all the three different aspects of μFUZZ help improve the bug detection capability, and the state partition contributes the most.

Code Coverage. As shown in Fig. 5, μFUZZ finds 23%, 7%, 6% more edge than μFUZZ-S, μFUZZ-C and μFUZZ-P respectively on average. We check μFUZZ-S's execution status and find that workers in the corpus management service of μFUZZ-S tend to select duplicated test cases for mutation. This is because without state partition the workers have the exact same state as each other and use the same scheduling algorithm. Such duplication can slow down fuzzers' exploration. Both μFUZZ-C and μFUZZ-P have a smaller fuzzing speed than μFUZZ (on different targets, 4% to 8% less), but for different reasons. For μFUZZ-C, it happens occasionally that some workers in a service have finished processing the requests and others have not. Even though we have available cores to run workers in other services, μFUZZ-C cannot do so without concurrency. For μFUZZ-P, it has to spend more computation power on copying data than μFUZZ.

Overall, μFUZZ outperforms μFUZZ-S, μFUZZ-C and μFUZZ-P in bug detection and code coverage, meaning that state partition, concurrency and fast communication all contribute to μFUZZ's improvement. State partition allows all the workers to work independently and still achieves a great aggregated result. The concurrent design allows μFUZZ to run differnet services concurrently without blocking. And the fast communication allows μFUZZ to reduce the overhead of data copying.



(a) PHP **(b)** `SQLite` **(c)** `Poppler`

(d) `sndfile` **(e)** `openssl` **(f)** `libxml2`

(g) `libpng` **(h)** `lua` **(i)** `libtiff`

Fig. 5: Edge coverage found by evaluated fuzzers with 40 cores in 24h. AFLpp-FS is `AFLplusplus` with 30-second synchronization interval. μFUZZ-S is μFUZZ without state partitioning. μFUZZ-P is μFUZZ without zero copy communication. μFUZZ-C is μFUZZ without concurrency.

5.4 Identified New Bugs

μFUZZ find 11 new bugs in four well-tested programs from `FuzzBench`, showing that μFUZZ is applicable in real-world fuzzing. We do not use `Magma` for new bug detection because `Magma` uses the fixed old version of the programs to insert bugs stably. The details for the bugs are shown in Table 4. The identified bugs include four logical errors (*i.e.*, assertion failure), six memory corruption errors(*e.g.*, heap use-after-free), and a memory leak error, of which nine have been fixed and the remainning two acknowledged by the developers at the time of paper acceptance.

6 Discussion

In this section, we present some limitations of the current implementation of μFUZZ and discuss their possible solutions.

6.1 Distributed Fuzzing

Currently, μFUZZ is implemented as a multithreaded program running in a single machine. μFUZZ can be extended to support distributed fuzzing in two ways. One way is to run one μFUZZ on each machine and perform state synchronization by connecting services with remote procedure calls (RPC). For example, we can run μFUZZ on different machines and connect their corpus management services to synchronize the corpus, which is the state-of-the-art approach. This will transfer the same amount of data across machines as existing approaches. However, existing serial fuzzers will pause to perform the slow network I/O, while μFUZZ allows other services to progress concurrently. If one service pauses, the other services can still run and make individual progress. Another way is to run different services on different machines and communicate over the network. This will greatly increase the amount of network data because the zero copy communication is not applicable. Although the data copy is inevitable, we can mitigate the waiting for network I/O. We can warm up the service by input caching: before a service starts, it fetches enough inputs from other services into its cache. Afterward, each service keeps fetching more inputs from other services into the cache and running the workers to consume the inputs concurrently. In this way, the service can keep running without waiting for inputs. Since each service will fetch inputs at demand, one straightforward way to achieve dynanmic load balancing is always picking the producer serivice with the most cached outputs. Suppose we have two test case generation services and two execution services and each of them runs on a different server with the same computation power. Each of the execution service should connect to both test case generation services. When the execution service needs to fetch more test cases, it picks the test case generation service with the more cached test cases to fetch from. We should make sure the amount of each fetch is not too large, which depends on the network capability, to avoid that one service fetches all the data and the other service is starved. Therefore, to support distributed fuzzing, μFUZZ might suffer from the

Table 4: New bugs detecte by μFUZZ.

Target	Type	Status	Ref
PHP (8.3.0-dev)	Use-After-Free	Fixed	https://github.com/php/php-src/issues/10582
	Use-After-Free	Fixed	https://github.com/php/php-src/issues/10169
	Use-After-Free	Acknowledged	https://github.com/php/php-src/issues/10581
	Use-After-Free	Acknowledged	https://github.com/php/php-src/issues/10571
	Null Pointer Deref	Fixed	https://github.com/php/php-src/issues/9543
	Memory Leak	Fixed	https://github.com/php/php-src/issues/10582
	Assertion Failure	Fixed	https://github.com/php/php-src/issues/10570
	Assertion Failure	Fixed	https://github.com/php/php-src/issues/10085
lua(5.4.4)	Assertion Failure	Fixed	http://lua-users.org/lists/lua-l/2023-02/msg00020.html
freetype(2.12.1)	Segmentation Fault	Fixed	https://gitlab.freedesktop.org/freetype/freetype/-/issues/1197
FFmpeg (9903ba)	Assertion Failure	Fixed	https://trac.ffmpeg.org/ticket/9908

same overhead as or more overhead than existing fuzzers due to data copying, but it can mitigate the blocking of network I/O by its concurrency.

6.2 Support More Mutation Strategy

Currently, μFUZZ implements some basic state-of-the-art fuzzing strategies such as `AFLplusplus`'s havoc mutation and edge coverage guidance. We can integrate advanced strategies into μFUZZ to futher improve μFUZZ's applicability. Stateless fuzzing strategies can be easily integrated thanks to μFUZZ's modularized design. For example, we can use advanced mutation strategies such as AST mutation [13, 83] as the workers in the mutation service. We can also use persistent executors [2, 6] or binary-only executors [11] in the execution service. For stateful ones, we should apply dynamtic or static partitioning to avoid as much synchronization as possible. To support CmpLog [14], μFUZZ can log the compared values in the execution service and pass the log to the feedback collection service. Then it categorizes the log and passes it to the corpus management service, which maintains the log as metadata of the test cases and passes them to guide the mutation service. Since the test cases are partitioned in the corpus management service, the compared values log is also partitioned and maintained independently. To support MOPT [46] mutators in μFUZZ, we partition different mutators into different workers in the mutation service. The feedback service can send back the information about the mutators' performance (e.g., the number of interesting inputs found by each mutator). Then mutation worker can adjust its local distribution and perform period synchronization with other workers to update the global optimal distribution. We see that some synchronization might be necessary. In this situation, the service might be blocked by locking, but the concurrency still allows other services to make progress. Therefore, one best-effort strategy is to adopt μFUZZ's architecture as much as possible, and fall back to state synchronization when necessary. We plan to integrate μFUZZ with `LibAFL` [30], an open-sourced fuzzing development kit that implements many advanced fuzzing strategies, including all the discussed ones, as self-contained reusable modules. For the stateless strategies, we can directly reuse modules in the workers as both μFUZZ and `LibAFL` are written in Rust. For the stateful ones, we need to first separate the data (*i.e.*, the fuzzing state) and operations in the module and then design the partitioning strategy, which requires nonnegligible engineering efforts. Luckily, μFUZZ takes care of data interaction and communication between the services, so the users can treat the data as local and focus on the partitioning strategy.

6.3 Support Collaborative Fuzzing

Currently μFUZZ has not implemented collaborative fuzzing [25, 55], which combines all kinds of different fuzzers to get a higher overall fuzzing performance. We can support collaborative fuzzing with μFUZZ by increasing the vairety of the workers (*i.e.*, using workers with different fuzzing strategies). For example, we can use both grammar-based mutation workers and bitflip mutation workers in the mutation service, and we can use execution workers with different type of instrumentation in the execution service. However, different fuzzing strategies might have different fuzzing states for the same functionality, which means that we should distinguish workers of different types of fuzzing states and dispatch the inputs accordingly. For example, grammar fuzzers might maintain a corpus in the form of abstract syntax trees (AST) instead of a binary stream. Suppose we use both grammar-based mutators and bit-level mutators in the mutation service. In that case, the input dispatcher of the test case generation service should dispatch inputs of AST test cases to a worker of grammar-based mutation instead of a worker of bitflip mutation. μFUZZ can support this by tagging both workers and data. Each worker has a tag to indicate the type of the inputs it can consume. Every result generated by a worker will be tagged to show which workers can consume it. With that, the input dispatcher can dispatch inputs to matching the inputs and workers. In this way, μFUZZ can combine fuzzing strategies with different types of fuzzing states.

7 Related Work

7.1 Fuzzing Strategy Improvement

Improving the fuzzing strategy focuses on enhancing the internal components of a fuzzer, including test case generation, feedback, and seed scheduling. There are mainly two types to test case generation in fuzzing: generation-based fuzzing [34, 49, 76, 77] and mutation-based fuzzing [47, 74, 79]. Generation-based fuzzing focuses on testing software that consumes structural inputs [1, 40, 49, 57, 68]. They typically utilize the grammar model of the inputs to generate structural inputs that can pass the format checks. SQLSmith [1] uses the SQL grammar and database schemes to generate valid queries. MoWF [57] leverages the file format information to fuzz the deeper program code beyond the parser. Mutation-based fuzzing performs mutation on existing test cases to generate new ones. In this way, the fuzzer can utilize feedback information from the execution phase to guide its mutation. AFL [79] uses edge coverage to model program states to guide its mutation, which is shown to be highly effective. Adopting the methodology from generation-based fuzzers, some language processor fuzzers [13, 24, 83] utilize language grammar to perform constrained mutation. Other fuzzers [20, 67, 78] use symbolic execution or concolic execution to get through complex program conditions. T-Fuzz [56] further proposes a way to dynamically transform the program in order to remove certain checks that are hard for the fuzzer to bypass successfully. To improve feedback quality, researchers try to find better models for the program states. CollAFL [32] provides more accurate coverage information by mitigating path collisions in AFL. Some fuzzers [14, 15, 22, 31, 33, 61] use taint analysis to incorporate data flow information into their coverage metrics. PATA [45] further proposes a path-aware taint analysis by distinguishing between multiple occurrences of the same constraint. The learning-enabled fuzzer NEUZZ [64] leverages a surrogate neural network to smoothly approximate the branching behavior of the program in order to generate useful test cases. Another way is to improve the seed scheduling algorithm [65, 81]. AFLFast [19], MOPT [18], DigFuzz [82] collect information about the test cases and prioritize those with higher potential to reach new code regions. μFUZZ does not improve existing fuzzing strategies but focuses on better parallelizing these strategies.

7.2 Fuzzing Speed Improvements

Improving fuzzing speed allows fuzzers to run more executions in the same amount of time with the same fuzzing strategy [21, 27, 39, 41, 52, 53, 62, 71], which is usually orthogonal to the fuzzing strategy. Various techniques [26, 52, 54, 71] have been proposed to improve the instrumentation of the target program to reduce its overhead. Nagy et al. [52, 54] proposes coverage-guided tracing to trace code coverage only when new ones are discovered. Odin [71] adopts dynamic recompilation to prune necessary instrumentation on the fly. RetroWrite [26] uses static binary rewriting to support high-speed coverage-guided binary-only fuzzing with an efficient binary-only Address Sanitizer. Researchers have also explored hardware-assisted feedback-collecting mechanisms. kAFL [63], Honggfuzz [35], and PTrix [23] utilize Intel's *Processor Trace* technology, which enables them to efficiently collect coverage feedback with minimum overhead. Another well-explored topic is to improve the symbolic execution speed for hybrid fuzzing. Qsym [78] implements a symbolic execution engine tailored for fuzzing. Instead of translating the instructions to the intermediate representation and then executing them symbolically, Qsym tightly integrates the symbolic emulation with the native execution. SymCC [59] generalizes the idea of Qsym and presents a compiler that builds concolic execution right into the binary. In this way, the symbolic execution engine can run natively without any interpretation. Furthermore, utilizing QEMU, SymQEMU [60] modifies the IR of the target program before it gets translated into the host architecture, which enables compiling symbolic execution capabilities into the binary without access to its source code. Efforts to improve the fuzzing speed can also be combined with μFUZZ to facilitate the parallel fuzzing performance.

7.3 Parallel Fuzzing

Existing works improve the performance of parallel fuzzing also by either improving the fuzzing strategy [25, 44, 58, 66, 72, 84] or improving the fuzzing speed [75]. One popular way to improve the fuzzing strategy is task partitioning. PAFL [44] proposes an efficient guiding information synchronization method and statically divides fuzzing tasks based on branching information to reduce the overlap between instances. AFLEdge [72] further utilizes static analysis to dynamically create mutually exclusive and evenly weighted fuzzing tasks. Another way to improve the fuzzing strategy is to combine the capabilities of different fuzzers, which is also called ensemble fuzzing [25] or collaborative fuzzing [37]. The main idea is that different fuzzers might have different strengths on different targets. We can fuzz the same target with different fuzzers and share their fuzzing progress to let them help each other and achieve an overall better performance. EnFuzz [25] designs three heuristics for evaluating the diversity of existing fuzzers and choosing the most diverse subset to perform ensemble fuzzing through efficient seed synchronization. Cupid [37] further proposes a collaborative fuzzing framework that can automatically discover the best combination of fuzzers for a target. One well-known problem of parallel fuzzing is the bottleneck of the underlying operating system. Xu et al. [75] found that the fuzzing performance can significantly degrade when running with multiple cores due to the file system contention and the scalability of the `fork` system call. Thus, they proposed three new operating primitives that allow much higher scalability and performance for parallel fuzzing. The current state-of-the-art fuzzers [2, 6, 35] support persistent fuzzing mode, which reuses the same process for multiple test cases to reduce the overhead of forking. Moreover, in-memory test cases [2] are also adopted to reduce the I/O overhead and file system contention.

8 Conclusion

We present μFUZZ, a parallel fuzzing framework with microservice architecture that supports concurrency to better utilize CPU power in the existence of blocking I/O and avoids state synchronization with state partition. Our evaluation shows μFUZZ is more effective in parallel fuzzing than existing fuzzers with 24% improvement in code coverage and 33% improvement in bug detection than the second-best fuzzer in 24 hours. Besides, μFUZZ finds 11 new bugs in well-tested real-world programs.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. The work was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2179, N00014-17-1-2895, N00014-15-1-2162, N00014-18-1-2662, N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894, the National Science Foundation (NSF) under grant CNS-1652790 and Cisco Systems under an unrestricted gift.

References

[1] SQLSmith. https://github.com/anse1/sqlsmith, 2016.

[2] Aflplusplus. https://github.com/AFLplusplus/AFLplusplus, 2022.

[3] Build reliable network applications without compromising speed. https://tokio.rs/, 2022.

[4] Clusterfuzz. https://github.com/google/clusterfuzz, 2022.

[5] Clusterfuzzlite. https://google.github.io/clusterfuzzlite/, 2022.

[6] libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2022.

[7] Microservice architecture. https://microservices.io/, 2022.

[8] Oss-fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2022.

[9] Quickjs javascript engine. https://bellard.org/quickjs/, 2022.

[10] A self-hosted fuzzing-as-a-service platform. https://github.com/microsoft/onefuzz, 2022.

[11] Important features of afl++. https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/features.md, 2023.

[12] The source code of readdir system call. https://elixir.bootlin.com/linux/v6.1.9/source/fs/readdir.c#L54, 2023.

[13] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[14] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[15] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.

[16] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1985–2002, USA, 2019. USENIX Association.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

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

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[21] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1531–1531. IEEE Computer Society, 2022.

[22] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[23] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 633–645, 2019.

[24] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 642–658. IEEE, 2021.

[25] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. {EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, 2019.

[26] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[27] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. Hardware support to improve fuzzing performance and precision. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2214–2228, 2021.

[28] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[29] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[30] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1051–1065, 2022.

[31] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594, 2020.

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

[33] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009.

[34] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008.

[35] Google. Honggfuzz, 2016. https://google.github.io/honggfuzz/.

[36] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Master thesis, TU Braunschweig*, 2018.

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

[38] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[39] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[40] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 259–269, New York, NY, USA, 2018. Association for

Computing Machinery.

[41] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[42] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben ten Hove, and Marcel Böhme. Effectiveness and scalability of fuzzing techniques in ci/cd pipelines. *arXiv preprint arXiv:2205.14964*, 2022.

[43] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

[44] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 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*, pages 809–814, 2018.

[45] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA*, pages 154–170, 2022.

[46] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[47] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss*, 2022.

[48] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[49] MozillaSecurity. funfuzz. https://github.com/MozillaSecurity/funfuzz, 2020.

[50] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. {MundoFuzz}: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1257–1274, 2022.

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

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

[53] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700, 2021.

[54] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 351–365, New York, NY, USA, 2021. Association for Computing Machinery.

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

[56] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018.

[57] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016.

[58] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1337–1341. IEEE, 2021.

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

[60] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[61] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[62] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2597–2614, 2021.

[63] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[64] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[65] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. *arXiv preprint arXiv:2203.12064*, 2022.

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

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[68] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[69] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[70] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *2021 Network and Distributed System Security Symposium*, 2021.

[71] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: On-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 1010–1024, New York, NY, USA, 2022. Association for Computing Machinery.

[72] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. Facilitating parallel fuzzing with mutually-exclusive task distribution. In *International Conference on Security and Privacy in Communication Systems*, pages 185–206. Springer, 2021.

[73] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming

Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the International Conference on Software Engineering*, 2022.

[74] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. Cooper: Testing the binding code of scripting languages with cooperative mutation. In *29th Annual Network and Distributed System Security Symposium.*, 2022.

[75] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[76] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, New York, NY, USA, 2011.

[77] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 435–450, 2021.

[78] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, USA, 2018.

[79] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, 2019.

[80] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[81] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. *arXiv preprint arXiv:2201.04441*, 2022.

[82] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[83] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, November 2020.

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

Table 5: Line of codes of different components of μFUZZ, which sum up to 9534 lines.

Module	Language	LOC
Concurrent Runtime	Rust	1,980
Corpus Management	Rust	759
Testcase Mutation	Rust	1,604
Fork-Server Execution	Rust	1,453
Feedback Collection	Rust	1,169
Others	Rust/Protobuf	2,569
Total	Rust/Protobuf	**9,534**