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 $\mu$Fuzz. To better utilize CPU power in the existence of I/O, $\mu$Fuzz breaks down the synchronous fuzzing loops into concurrent microservices, each with multiple workers. To avoid state synchronization, $\mu$Fuzz 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 $\mu$Fuzz outperforms the second-best existing fuzzers with 24% improvements in code coverage and 33% improvements in bug detection on average in 24 hours. Besides, $\mu$Fuzz 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 gen-
eration 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 will release the code of µFUZZ upon publication.

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.

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.

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:

![Flowchart of AFLplusplus](image)

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

<table>
<thead>
<tr>
<th>Target</th>
<th>lua</th>
<th>PHP</th>
<th>tcpdump</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wasted Cycles (%)</td>
<td>1.2%</td>
<td>0.3%</td>
<td>28.7%</td>
<td>60.3%</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Interval/Instance</th>
<th>1 sec</th>
<th>1 min</th>
<th>30 min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>lua</td>
<td>PHP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Wasted Cycles (%)</td>
<td>3.18%</td>
<td>0.13%</td>
<td>0.07%</td>
</tr>
</tbody>
</table>

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

![Diagram](image)
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 further 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 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 suboptimal 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 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.

Fig. 3 shows the overview of \(\mu\text{FUZZ}\). 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 of the test cases, the scheduling algorithm prioritizes those that 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.

**Concentration 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 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 synchronizing 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 "interestingness" 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 service 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 is 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

---

Algorithm 1: State Partition & Update

```
1 Procedure StatePartition(state)
2    numWorkers ← The number of workers in the service;
3    if state is fixed-sized then // Static partition
4        partitions ← Evenly partition the state based on numWorkers;
5    else
6        state ← RandomlyShuffle(state); // Dynamic partition
7        partitions ← Evenly partition the state based on numWorkers;
8    Distribute one partition to one worker;
9    The worker tags the partitioned state with its ID and store it
10 Procedure StateUpdate(stateUpdateRequest)
11    id2WorkerMap ← Map(ID, worker);
12    id ← ExtractID(stateUpdateRequest);
13    worker ← id2WorkerMap.Get(id);
14    worker.ProcessRequest(stateUpdateRequest)
```
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 according 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 fork-server 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)
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 $\infty$. 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 opens1 and PHP because no fuzzers find any bugs in these targets in 24 hours.

<table>
<thead>
<tr>
<th>Targets</th>
<th>Bug ID</th>
<th>$\mu$Fuzz</th>
<th>AFLplusplus</th>
<th>AFLEdge</th>
<th>AFLTeam</th>
<th>AFLpp-FS</th>
<th>$\mu$Fuzz-S</th>
<th>$\mu$Fuzz-P</th>
<th>$\mu$Fuzz-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poppler</td>
<td>PDF010</td>
<td>12h53m</td>
<td>19h18m</td>
<td>22h13m</td>
<td>03h13m</td>
<td>13h13m</td>
<td>$\infty$</td>
<td>15h02m</td>
<td>17h40m</td>
</tr>
<tr>
<td></td>
<td>PDF016</td>
<td>01m40s</td>
<td>16m40s</td>
<td>01m40s</td>
<td>01m40s</td>
<td>01m40s</td>
<td>$\infty$</td>
<td>01m40s</td>
<td>09m10s</td>
</tr>
<tr>
<td></td>
<td>PDF021</td>
<td>05h22m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>13h51m</td>
</tr>
<tr>
<td>sndfile</td>
<td>SNDO17</td>
<td>01h24m</td>
<td>03h34m</td>
<td>02h51m</td>
<td>02h46m</td>
<td>16m45s</td>
<td>02h46m</td>
<td>03h54m</td>
<td>03h40m</td>
</tr>
<tr>
<td></td>
<td>SNDO20</td>
<td>01h24m</td>
<td>04h15m</td>
<td>02h51m</td>
<td>02h46m</td>
<td>02h50m</td>
<td>02h46m</td>
<td>04h10m</td>
<td>02h54m</td>
</tr>
<tr>
<td>XML002</td>
<td></td>
<td>12h57m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>14h31m</td>
</tr>
<tr>
<td>XML003</td>
<td></td>
<td>02h46m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>13h53m</td>
</tr>
<tr>
<td>libxml2</td>
<td>XM009</td>
<td>16m40s</td>
<td>02h46m</td>
<td>02h46m</td>
<td>02h46m</td>
<td>02h46m</td>
<td>02h46m</td>
<td>02h46m</td>
<td>23m53s</td>
</tr>
<tr>
<td>XML012</td>
<td></td>
<td>15h13m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>20h45m</td>
<td>20h07m</td>
<td>17h02m</td>
<td>19h49m</td>
</tr>
<tr>
<td>XML017</td>
<td></td>
<td>01m40s</td>
<td>16m40s</td>
<td>01m40s</td>
<td>01m40s</td>
<td>01m40s</td>
<td>02h48m</td>
<td>06m40s</td>
<td>01h53m</td>
</tr>
<tr>
<td>SQLite</td>
<td>SQL002</td>
<td>02h46m</td>
<td>10h34m</td>
<td>07h28m</td>
<td>$\infty$</td>
<td>03h16m</td>
<td>06h19m</td>
<td>17h23m</td>
<td>03h12m</td>
</tr>
<tr>
<td></td>
<td>SQL018</td>
<td>02h46m</td>
<td>02h52m</td>
<td>05h14m</td>
<td>$\infty$</td>
<td>03h40m</td>
<td>03h33m</td>
<td>05h16m</td>
<td>05h27m</td>
</tr>
<tr>
<td>lua</td>
<td>LUA004</td>
<td>02h47m</td>
<td>09h47m</td>
<td>19h11m</td>
<td>06h20m</td>
<td>16h40m</td>
<td>19h48m</td>
<td>02h48m</td>
<td>02h55m</td>
</tr>
<tr>
<td>PNG001</td>
<td></td>
<td>19h37m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>libpng</td>
<td>PNG003</td>
<td>01m40s</td>
<td>03m25s</td>
<td>01m40s</td>
<td>02m20s</td>
<td>01m55s</td>
<td>17m20s</td>
<td>17m30s</td>
<td>$\infty$</td>
</tr>
<tr>
<td>PNG007</td>
<td></td>
<td>02h28m</td>
<td>02h50m</td>
<td>21h32m</td>
<td>23h05m</td>
<td>18h36m</td>
<td>06h03m</td>
<td>02h46m</td>
<td>03h03m</td>
</tr>
<tr>
<td>TIFF002</td>
<td></td>
<td>05h56m</td>
<td>$\infty$</td>
<td>18h17m</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>12h32m</td>
<td>$\infty$</td>
</tr>
<tr>
<td>libtiff</td>
<td>TIFF007</td>
<td>01m40s</td>
<td>16m40s</td>
<td>05h53m</td>
<td>01m40s</td>
<td>02h47m</td>
<td>05h34m</td>
<td>02h46m</td>
<td>02h51m</td>
</tr>
<tr>
<td>TIFF012</td>
<td></td>
<td>02h35m</td>
<td>09h26m</td>
<td>06h23m</td>
<td>02h46m</td>
<td>10h35m</td>
<td>14h29m</td>
<td>06h39m</td>
<td>02h55m</td>
</tr>
<tr>
<td>TIFF014</td>
<td></td>
<td>05h47m</td>
<td>03h11m</td>
<td>06h06m</td>
<td>04h22m</td>
<td>02h50m</td>
<td>12h06m</td>
<td>04h44m</td>
<td>03h01m</td>
</tr>
<tr>
<td>Total Bugs Found</td>
<td>20</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>18</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

5.1 Evaluation Setup

Benchmark. We use the state-of-the-art benchmark Magma [38] to evaluate $\mu$Fuzz 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 $\mu$Fuzz 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 $\mu$Fuzz’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 $\mu$Fuzz 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 $\mu$Fuzz, 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 $\mu$Fuzz 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, $\mu$Fuzz 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 $\mu$Fuzz, and $\mu$Fuzz found 12 of them using the shortest time. Three of the bugs (PDF021, XML002, XML003) are only found by $\mu$Fuzz. Additionally, AFLpp-FS finds all the 14 bugs found by AFLplusplus and one more, but it still finds five less than $\mu$Fuzz. 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 \( \muFuzz \) 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, \( \muFuzz \) can achieve higher code coverage improvement (e.g., 40% more in Poppler and 37% more in PHP than AFLplusplus). Otherwise, the improvement of \( \muFuzz \) is smaller (8% more in sndfile than AFLplusplus and 2% less than AFLpp-FS). \( \muFuzz \) 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 \( \muFuzz \). 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 comparison, \( \muFuzz \) updates its state continuously and timely, and thus avoids using obsolete states. Although \( \muFuzz \)’s partition is not as comprehensive as that of AFLTeam and AFLEdge, the timely state updates allow \( \muFuzz \) to make fuzzing decisions that fit the current state.

Overall, \( \muFuzz \) outperforms the four compared fuzzers in both bug detection and code coverage under parallel fuzzing. The fuzzing effectiveness of \( \muFuzz \) comes from both its architecture and state partition.
5.3 Contribution of Components

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

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

**Code Coverage.** As shown in Fig.5, \(\mu\text{Fuzz}\) finds 23%, 7%, 6% more edge than \(\mu\text{Fuzz-S}\), \(\mu\text{Fuzz-C}\) and \(\mu\text{Fuzz-P}\) respectively on average. We check \(\mu\text{Fuzz-S}\)’s execution status and find that workers in the corpus management service of \(\mu\text{Fuzz-S}\) 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 \(\mu\text{Fuzz-C}\) and \(\mu\text{Fuzz-P}\) have a smaller fuzzing speed than \(\mu\text{Fuzz}\) (on different targets, 4% to 8% less), but for different reasons. For \(\mu\text{Fuzz-C}\), 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, \(\mu\text{Fuzz-C}\) cannot do so without concurrency. For \(\mu\text{Fuzz-P}\), it has to spend more computation power on copying data than \(\mu\text{Fuzz}\).

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

5.4 Identified New Bugs

\(\mu\text{Fuzz}\) find 11 new bugs in four well-tested programs from FuzzBench, showing that \(\mu\text{Fuzz}\) 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.

<table>
<thead>
<tr>
<th>Target</th>
<th>Type</th>
<th>Status</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-After-Free</td>
<td>Fixed</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Use-After-Free</td>
<td>Acknowledged</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Use-After-Free</td>
<td>Acknowledged</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Use-After-Free</td>
<td>Acknowledged</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Null Pointer Deref</td>
<td>Fixed</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Memory Leak</td>
<td>Acknowledged</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Assertion Failure</td>
<td>Fixed</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>Assertion Failure</td>
<td>Acknowledged</td>
<td>***</td>
<td></td>
</tr>
</tbody>
</table>

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 six have been fixed and the remaining five acknowledged by the developers at the time of writing. We currently omit the references to the bug reports for anonymity.

6 Discussion

In this section, we present some limitations of the current implementation of \(\mu\text{Fuzz}\) and discuss their possible solutions.

6.1 Distributed Fuzzing

Currently, \(\mu\text{Fuzz}\) is implemented as a multithreaded program running in a single machine. \(\mu\text{Fuzz}\) can be extended to support distributed fuzzing in two ways. One way is to run one \(\mu\text{Fuzz}\) on each machine and perform state synchronization by connecting services with remote procedure calls (RPC). For example, we can run \(\mu\text{Fuzz}\) 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 \(\mu\text{Fuzz}\) 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 dynamic load balancing is always picking the producer service 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, $\mu$FUZZ might suffer from the 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, $\mu$FUZZ 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 $\mu$FUZZ to further improve $\mu$FUZZ’s applicability. Stateless fuzzing strategies can be easily integrated thanks to $\mu$FUZZ’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 dynamic or static partitioning to avoid as much synchronization as possible. To support CmpLog [14], $\mu$FUZZ 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 $\mu$FUZZ, 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 $\mu$FUZZ’s architecture as much as possible, and fall back to state synchronization when necessary. We plan to integrate $\mu$FUZZ 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 $\mu$FUZZ 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, $\mu$FUZZ 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 $\mu$FUZZ 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 $\mu$FUZZ by increasing the variety 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 fuzzer 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. $\mu$FUZZ 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, $\mu$FUZZ 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 fuzzers 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], DiFuzz [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 execution 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] propose 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 minimal 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.
References


Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming


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


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

<table>
<thead>
<tr>
<th>Module</th>
<th>Language</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Runtime</td>
<td>Rust</td>
<td>1,980</td>
</tr>
<tr>
<td>Corpus Management</td>
<td>Rust</td>
<td>759</td>
</tr>
<tr>
<td>Testcase Mutation</td>
<td>Rust</td>
<td>1,604</td>
</tr>
<tr>
<td>Fork-Server Execution</td>
<td>Rust</td>
<td>1,453</td>
</tr>
<tr>
<td>Feedback Collection</td>
<td>Rust</td>
<td>1,169</td>
</tr>
<tr>
<td>Others</td>
<td>Rust/Protobuf</td>
<td>2,569</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>Rust/Protobuf</td>
<td><strong>9,534</strong></td>
</tr>
</tbody>
</table>