# $\mu$FUZZ: Redesign Parallel Fuzzing using Microservice Architecture

*Anonymous*

## 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 use up-to-date information to make globally optimal decisions. Our experiments show that $\mu$FUZZ outperforms existing fuzzers with 57% improvements in code coverage and 67% improvements in bug detection on average in 24 hours. Besides, $\mu$FUZZ finds three new bugs in well-tested real-world programs.

## 1 Introduction

In recent years, fuzzing has been widely adopted as a software-testing technique to detect security bugs [17, 36, 76, 79]. Compared with other program analysis techniques, fuzzing ensures high 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 a set of optimizations to enhance each internal component [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 [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. As fuzzing shows its ability to detect bugs in complicated real-world software, many companies decide to invest a large number of resources such as CPU and memory in fuzzing [4, 11]. For example, Google deploys a parallel fuzz infrastructure to conduct CI/CD testing for newly submitted code [5]. The goal of parallel fuzzing is to make full use of resources and detect more bugs within a shorter time than single-instance fuzzing. State-of-the-art parallel fuzzing approaches share a similar architecture [2, 58, 72, 79]. Specifically, they launch multiple fuzzing instances in separate processes and periodically perform corpus synchronization from each other. 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 average speed of test case execution. The advantage of this parallel-fuzzing architecture comes from the synchronization, which allows one instance to catch up on the latest progress from other instances, so all instances contribute to the program state exploration for detecting bugs.

However, we identify two limitations that hinder scalability 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, for example, by file reading and writing, the whole instance gets stuck. The CPU bound to the fuzzing instance will be idle until the blocking operation completes. As parallel fuzzing will run multiple instances at the same time and introduces more I/O by synchronization, the instances are more likely to get stuck, resulting more CPU power wasted. We should reuse the idle CPU to fully utilize the computation power. A straightforward solution is to launch more instances such that the idle CPU can be switched to handle other non-blocking instances, like launching 32 instances on a 16-core machine. However, this method will introduce frequent context switches between different fuzzing instances, which again decreases the fuzzing performance.

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 optimal choices. 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 optimal 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 synchronization and serialization. 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 resource (i.e., CPU) 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 $\mu$FUZZ, a parallel fuzzing framework using the microservice architecture. In order 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. This effectively addresses the CPU idling issue since each microservice is loosely coupled and can replace the blocked service for execution. We take special care to allow one service to complete its task as much as possible before switching to others to minimize the overhead of context switches. We further design an output cache mechanism to reduce the coupling between different services, like 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.

To address the challenges caused by synchronization delay, we design two levels of state partition in $\mu$FUZZ. First, $\mu$FUZZ splits the global state based on microservices so that each service can use the 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 after executing test cases. Second, different workers in each service handle unique parts of the service states. Accumulating all worker states will obtain the service states. These partitions eliminate the state synchronization among different services and workers and enable each service worker to use up-to-date information to make globally optimal decisions.

We implement $\mu$FUZZ in 9534 lines of Rust code, which consists 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-4.01c`, 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 $\mu$FUZZ on popular benchmarks, including Magma [38] and FuzzBench [48]. We compared $\mu$FUZZ with the state-of-the-art parallel fuzzers, including `AFLplusplus`, `AFLEdge` and `AFLTeam`, and found $\mu$FUZZ can explore 57% more program states and 67% more bugs on average in 24 hours. Besides, our experiments show that $\mu$FUZZ can make progress in the existence of blocking I/O with its concurrent design, and its state partition helps improve code coverage by 96% and bug detection by 47%. Moreover, we evaluated $\mu$FUZZ and found three new bugs on well-tested real-world programs.

In summary, this paper makes the following contributions:

- We propose a novel parallel fuzzing framework with microservice architecture that well utilizes CPU power even with blocking I/O and eliminates the state synchronization.
- We implement the prototype, $\mu$FUZZ, of our system to effectively perform parallel fuzzing.
- We compared $\mu$FUZZ with state-of-the-art fuzzers, and the results show that $\mu$FUZZ can find 57% more new coverage and 67% more bugs in 24 hours on average.

We will release the code of $\mu$FUZZ upon publication.

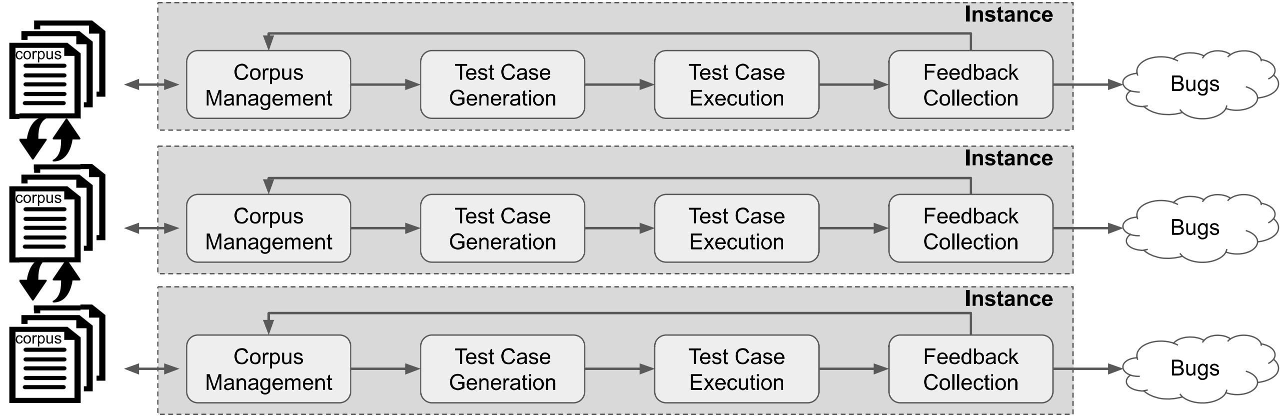


**Fig. 1: The state-of-the-art parallel fuzzing approach.** The fuzzer spawns multiple instances and runs them in parallel. Each instance is self-contained and functionality-complete. They maintain their own local fuzzing states, such as the corpus and coverage bitmap. Most of the time, the instances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

## 2 Problem

In this section, we first briefly describe how state-of-the-art parallel fuzzing approaches work. Next, we discuss the limitations of existing approaches. We then 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. Additionally, 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 each other's 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. The fuzzing state of a fuzzing instance is 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: (1). It aggravates the problem of CPU idling due to the serial design of the underlying fuzzer. (2). The global fuzzing state cannot be synchronized to each instance both timely and efficiently, which results in suboptimal performance.

**CPU Idling 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 I/O. I/O can come from two sources. First, the tested program involves 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 tasks, such as test case mutation, but wait, resulting in the CPU idling. For example, we measured the CPU usage of fuzzing `tcpdump` with `AFLplusplus` and found that the CPU usage was only 70%. We checked the system calls made by `tcpdump` using `strace` [12] and found that `tcpdump` was waiting for blocking system calls such as `poll` to return. Second, when fuzzing with multiple instances, state synchronization might also bring in lots of file I/O. 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]. A simple solution is to spawn more instances on the same CPU (*i.e.*, oversubscription). However, that might lead to higher CPU usage but

not better fuzzing performance. This is because, without a well-calibrated task scheduler, multiple instances will try to occupy the same CPU and cause lots of resource contention, such as excessive context switching. Such contention hurts the performance of each fuzzer and might result in a worse overall 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 the locally optimal fuzzing strategy, which can be suboptimal globally. On the other hand, we cannot synchronize too frequently, which has a high overhead [75].

We did a quick experiment to verify our hypothesis. We used `AFLplusplus` to fuzz `QuickJS` [9], a popular JavaScript engine and a benchmark from OSS-Fuzz [8]. 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 edge coverage and how many of the interesting test cases in the corpus 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 the coverage 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 thus 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 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 power even in the existence of I/O and can synchronize instances' states timely with little overhead so that we can use optimal fuzzing strategy. However, it is difficult to do so on top of existing fuzzers with monolithic serial architecture. We need a different architecture.

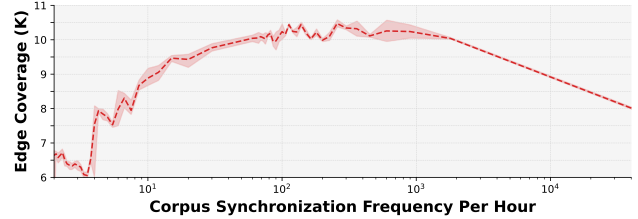


**Fig. 2: Code coverage of 10 `AFLplusplus` instances testing `QuickJS` with different synchronization frequency in an hour.**

## 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 complete functionality of fuzzing, where 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 fuzzing state it 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 globally optimal 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 optimal performance. We achieve our goal in two steps: redesigning the fuzzing framework with microservice architecture migration and partitioning the fuzzing state. Microservice architecture adds concurrency to the framework and enables the fuzzer to fully 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 optimal 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

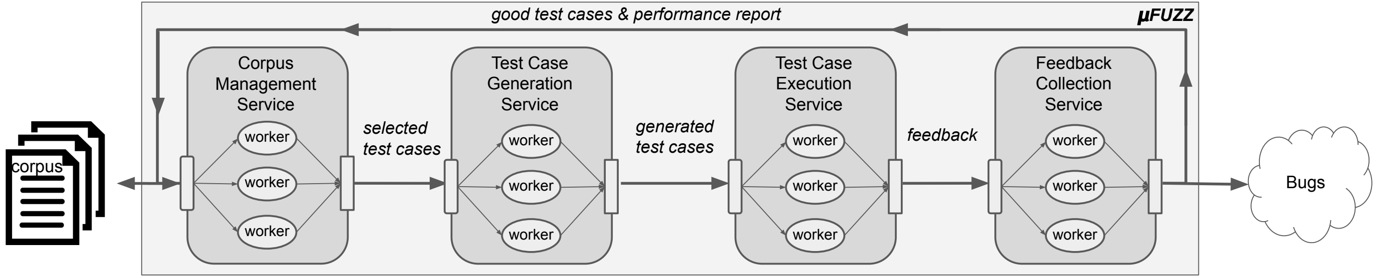


**Fig. 3:** Overview of μFUZZ. Instead of running multiple fuzzing instances and performing periodic synchronization, μFUZZ breaks the traditional monolithic architecture into a microservice one. The new architecture consists of four self-contained services, each maintaining a partition of the fuzzing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

is also partitioned into 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 dependent on each other: 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 maintain its own fuzzing state. However, if the state is shared by the workers, we still need state synchronization among the workers. We further partition the state among the workers inside the service to avoid synchronization. We use two rules to guide the partition. First, each partition of the state should be functionality-complete, which means the worker can finish its job without using others' states. Second, if each worker adopts its locally optimal strategy, we expect to get a globally optimal result by accumulating individual results. 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 optimal decisions. This avoids the problem of periodic synchronization, which either brings 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 into 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

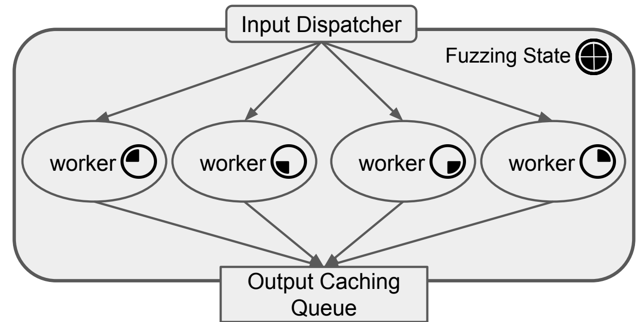


**Fig. 4:** The internal structure of a service in μFUZZ. Each service has a fuzzing state, an input queue dispatcher, an output caching queue, and some workers. The fuzzing state is partitioned into the workers. The input queue dispatcher accepts from other services and dispatches them to the workers.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

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 optimal 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 and reduce CPU idling due to blocking I/O, we break the monolithic serial fuzzing loop into multiple services. We use the following guidelines from the microservice architecture for the breakdown. First, each service should be micro and focus only on one core functionality of the fuzzing. 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 those finding new code coverage and those that trigger new bugs, etc. Based on the metadata of the test cases, the scheduling algorithm chooses some from the corpus that can help 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 (e.g., the code coverage information, whether the tested program crashes or timeouts during the execution, etc.).

**Feedback Collection Service.** It collects the feedback from the execution service and classifies whether they are 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 good 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. Therefore, these services still need to run in a serial way that one getting stuck blocks the overall progress. We need to further loosen the coupling between the services so that they can run concurrently and mitigate CPU idling, 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. 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.** One problem of output caching is that it allows the producer to generate unlimited outputs without constraints. Therefore, it might happen that one service keeps generating outputs and fully occupies 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 and be hung up 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 $\mu$FUZZ can have multiple workers in parallel. To achieve maximum parallelism, the number of workers should be the same as 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 partitioning. We define a worker as idle if it is ready to process input but not currently processing any. Such workers notify the input dispatcher to put them into the back of the queue in order. Whenever an input arrives, the input dispatcher tries to pop an idle worker out 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 partition before dispatching. If the arrived input is larger than a threshold value and there are more than one idle workers, we partition the input evenly based on the number of idle workers and dispatch each partition to an idle worker. With the two strategies, we can achieve maximum parallelism by keeping the workload of each worker balanced dynamically.

## 3.4 Avoid Synchronization by State Partition

As the result in Fig. 2 shows, if the fuzzing instances maintain their local states and rely on periodic synchronization, we end up with suboptimal performance due to either syn-

chronization lagging or high overhead. Therefore, we avoid synchronization by partitioning the fuzzing state. Afterward, we maintain only one global state but in a distributive way.

We already perform the first level of state partition by breaking the fuzzing loop into microservices. We further perform the second level inside each service. More specifically, we partition the state among the workers in a way that each worker maintains a unique and functionality-complete partition. Every worker only needs its own part of the state to finish its job. The workers do not need state synchronization with each other. Besides, the partition enables us to achieve an overall optimal result by simply accumulating the results from the workers. In this way, the workers can run independently.

We categorize the fuzzing states into two types. The first type has a known fixed size before the fuzzing starts. For example, the feedback collection service maintains a bitmap for recording code coverage. The size of the bitmap is fixed and configured by the users. The second type has an unknown variable size. For example, the seed corpus is part of the fuzzing state and grows as we fuzz. Due to these different characteristics, we perform static partition for the first type and dynamic partition for the second, as described below.

**Static State Partition.** For the fuzzing state of fixed size, we partition it statically and evenly among the workers. Therefore, we know which worker maintains which partition in advance. When new inputs arrive, the input dispatcher partition the inputs based on the partition boundary and dispatch them to the corresponding workers. 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 maintains byte 0 to byte 99, and the second maintains byte 100 to 199). When the execution service generates new bitmap information, the information is partitioned and dispatched to different workers in the feedback collection service. By accumulation, if none of the workers in the feedback collection service find new bits, the executed test case is considered uninteresting and otherwise interesting.

**Dynamic State Partition with Tagging.** For the fuzzing state of variable size, we cannot predict the size in advance to perform static partition. Instead, we use dynamic partition: whenever a new part of the state is generated, we partition it evenly and distribute it to the workers. Since the new state is randomly distributed, the partition in each worker should have similar data distribution as the global state. This is important because most fuzzing strategies are randomized ones, which means their performance is not dependent on the size or the specific values of the underlying data but on their distribution. Therefore, a fuzzing strategy that is optimal for the global state should also be optimal for the partitioned state. For example, if the corpus management service has 25 workers and maintains a corpus of 1000 interesting test cases. 500 of the test cases are considered as good (*i.e.*, they can potentially trigger more new codes), and the other 500 are bad. By uniformly random partition, each worker is expected to maintain 20 good test cases and 20 bad test cases. If the test case selection strategy for the global state is to first explore the good test cases and then the bad ones, then every worker just independently follows this strategy to achieve the same expected result.

One problem with dynamic partition is that there is no partition boundary. It is difficult for the input dispatcher to figure out how to dispatch the inputs. For example, the feedback collection service can send some performance reports of the evaluated test cases to the corpus management (e.g., the test case X generates 20 new test cases that trigger new code coverage) for fine-tuning the scheduling algorithm. The input dispatcher of the corpus management service cannot figure out which worker maintains the test case X and should receive such inputs easily. If we maintain the global knowledge of which worker maintains which test cases, it will be too much overhead as the corpus size grows. Therefore, for dynamic state partition, we assign each worker a unique ID and tag the states it maintains with the ID. Such tagged IDs will remain in all intermediate outputs that are related to the states (*i.e.*, the test cases sent out by the corpus management will contain the IDs of the workers maintaining them.). Since the number of workers can be known in advance, the input dispatcher can dispatch the inputs based on the ID.

## 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 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 only need eight bytes on an x64 system.

**Pointer Passing with Shared Memory.** Instead of asking both the producer and consumer service to unnecessarily allocate memory to store and copy the data from one to the other, we create shared memory between the services and pass the pointers to the shared memory. After the shared memory is set up, the producer service writes its outputs directly to the shared memory. To "pass" the data to the consumer, the pro-

**Table 1: Line of codes of different components of $\mu$FUZZ, 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** |

ducer 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.** Introducing shared memory incurs a potential safety problem. Since the shared memory is accessible from multiple services, if we allow the services to access the memory at the same time, race condition could happen. Therefore, we wrap the pointers with unique ownership to ensure safe memory access. This unique ownership guarantees that, at most one service can access the underlying shared memory at any moment. This is reasonable because the consumer should only access the output after the producer has finished generating it, and the producer has no need to access its output afterward.

## 4 Implementation

We implement $\mu$FUZZ in 9534 lines of code. Table 1 shows the breakdown.

**Concurrent Runtime.** We use `Tokio` as the concurrent runtime of $\mu$FUZZ. The runtime is responsible for efficient task scheduling. Each worker in the services of $\mu$FUZZ is run as a task in the runtime. As users configure the number of workers, the number of total tasks is fixed. In this way, we avoid the overhead of unnecessary task creation. We maintain a double-ended 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 $\mu$FUZZ will try to put them 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. To avoid unnecessary service switching, when a service receives inputs, it processes as many of them as possible. If all the inputs are processed or the service gets stuck, $\mu$FUZZ 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.

**Table 2: Line of source codes and the number of inserted bugs of the tested six targets.** Larger code spaces usually result in more complex programs.

| Target | `Poppler` | `SQLite` | `openssl` | `sndfile` | `libxml2` | `PHP` |
|---|---|---|---|---|---|---|
| **LOC** | 342K | 320K | 695K | 66K | 457K | 1,488K |
| **Bug Num** | 22 | 20 | 20 | 18 | 17 | 16 |

**Test Case Generation.** $\mu$FUZZ 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, considering the fuzzing speed. Instead, we mutate each test case multiple times and send the new variants in bulk to the execution service to reduce the overhead. We also skip the deterministic stage and only perform the havoc stage as `AFLplusplus`'s parallel fuzzing mode does.

**Execution.** The execution service adopts the popular fork-server approach [2, 79]. Each worker in the execution service has its own fork server. When a worker receives an input to execute, it feeds the input into the fork server and requests a 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 $\mu$FUZZ in the same process to share the address space. 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, which means only one owner can operate on the underlying memory.

## 5 Evaluation

Our evaluation aims to answer the following questions.

- Can $\mu$FUZZ outperform state-of-the-art parallel fuzzers? (§5.2)
- Can $\mu$FUZZ's microservice architecture and state partition improve fuzzing performance?
- Can $\mu$FUZZ 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 $\mu$FUZZ. The measured metrics in-

clude bug detection capability and code coverage. Due to the time and resource limit, we rank the programs in Magma by the number of inserted bugs and test the top six programs: Poppler, SQLite, openssl, sndfile,libxml2, PHP. We use the corpus from Magma for all the targets and run them through AFLplusplus's test case minimizers to remove redundant ones. 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 still 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 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.

**Environment Setup.** We perform our evaluation on three machines, each with an Ubuntu 18.04 operating system, an Intel Xeon CPU E5-2680 v3 processor with 48 visual cores and 256GB memory. For bug detection, we calculate the number of bugs with Magma, which assigns a unique bug ID to all its inserted bugs and prints a log whenever a bug is triggered. Due to the randomness in fuzzing, we further apply Magma's survival analysis to convert bug triggering time to bug survival time, which is the expected time a bug remains undiscovered. A smaller survival time indicates a fuzzer can find the bug in shorter time. We instrument the tested programs to test edge coverage with hit counters. For the code coverage and bug detection experiments, we run the fuzzers with 40 fuzzing instances on 40 cores inside docker for 24 hours and repeat the process five times. For μFUZZ, we run 40 workers for each service but still use only 40 cores which is the same as the other fuzzers. We 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 to understand its strengths and weaknesses in parallel fuzzing, including the *de facto* AFLplusplus and the two most recent parallel fuzzers, AFLEdge and AFLTeam. Since μFUZZ's seed scheduling algorithm is different from AFLplusplus's, we also compare with AFLplusplus-M, which is AFLplusplus with μFUZZ's seed scheduling algorithm, to see whether performance improvement is due to our seed scheduling. We also want to understand whether increasing the synchronization frequency of existing fuzzers can improve fuzzing as a simple solution. Therefore, we add a comparison with AFLplusplus-F, which is AFLplusplus performing synchronization every 30 seconds instead of 30 minutes. We use 30 seconds because we find by experiments that it has lower overhead and achieves almost the same coverage as that of a shorter synchronization interval. We compare these fuzzers in two metrics: bug detection (the number of triggered bugs and their survival time) and edge coverage.

**Bug Detection.** As shown in Table 3, μFUZZ finds 12 bugs in 24 hours, while AFLplusplus, AFLEdge, AFLTeam, AFLplusplus-M, and AFLplusplus-F find only 8, 8, 6, 7, and 9 bugs, respectively. All the nine bugs found by other fuzzers are also covered by μFUZZ, and μFUZZ found seven of them within the shortest time. Three of the bugs (PDF201, XML002, XML003) are only found by μFUZZ. AFLplusplus finds one more bug than AFLplusplus-M, meaning that the seed scheduling of μFUZZ might not be as good as AFLplusplus's in bug detection. However, μFUZZ can still find more bugs quickly, thanks to the microservice design and state partition. We plan to adopt the advanced seed scheduling algorithm of AFLplusplus to further improve the performance of μFUZZ. Additionally, AFLplusplus-F finds all the eight bugs found by AFLplusplus and one more. Out of the 8 common bugs, AFLplusplus-F finds six of them faster. This shows that higher synchronization does help improve the bug detection capability of AFLplusplus because the fuzzing instances can catch up with the latest progress earlier.

**Code Coverage.** As shown in Fig. 5, on average μFUZZ identifies 23%, 45%, 102%, 88%, and 33% more new edges than AFLplusplus, AFLEdge, AFLTeam, AFLplusplus-M, and AFLplusplus-F respectively on the six targets. We can see from Table 2 that the improvement of μFUZZ against the second-best fuzzer is related to the complexity of the targets. If the programs have larger code bases and more program states to explore , μFUZZ can achieve much higher code coverage than the second-best fuzzer (e.g., 40% more in Poppler and 37% more in PHP than AFLplusplus). Otherwise, if the programs are small and fuzzers can achieve saturated coverage in a short time, then the improvement of μFUZZ is not apparent (10% more in openssl than AFLplusplus and 6% less in sndfile than AFLplusplus-F). Another thing we see is that AFLplusplus has higher code coverage than AFLplusplus-M, which shows that the simple scheduling algorithm of μFUZZ is not as good as AFLplusplus's in input space exploration. This means the improvement of μFUZZ does not come from its scheduling algorithm but its new architecture and state partition. We notice that AFLEdge and AFLTeam have worse performance than AFLplusplus. We carefully investigated their source code and execution status and found that they both run their partition algorithms at a fixed time interval (i.e., every hour). The algorithms aggregate the fuzzing progress of all their instances (e.g., the corpus of all instances) and then perform partitioning, which requires heavy analysis. However, since we are running the experiment with 40 instances and the total size of the corpus is large, the analysis can take 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 and the partition results might not be optimal anymore. Interestingly, we notice that the coverage of AFLplusplus-F is higher than AFLplusplus at the beginning

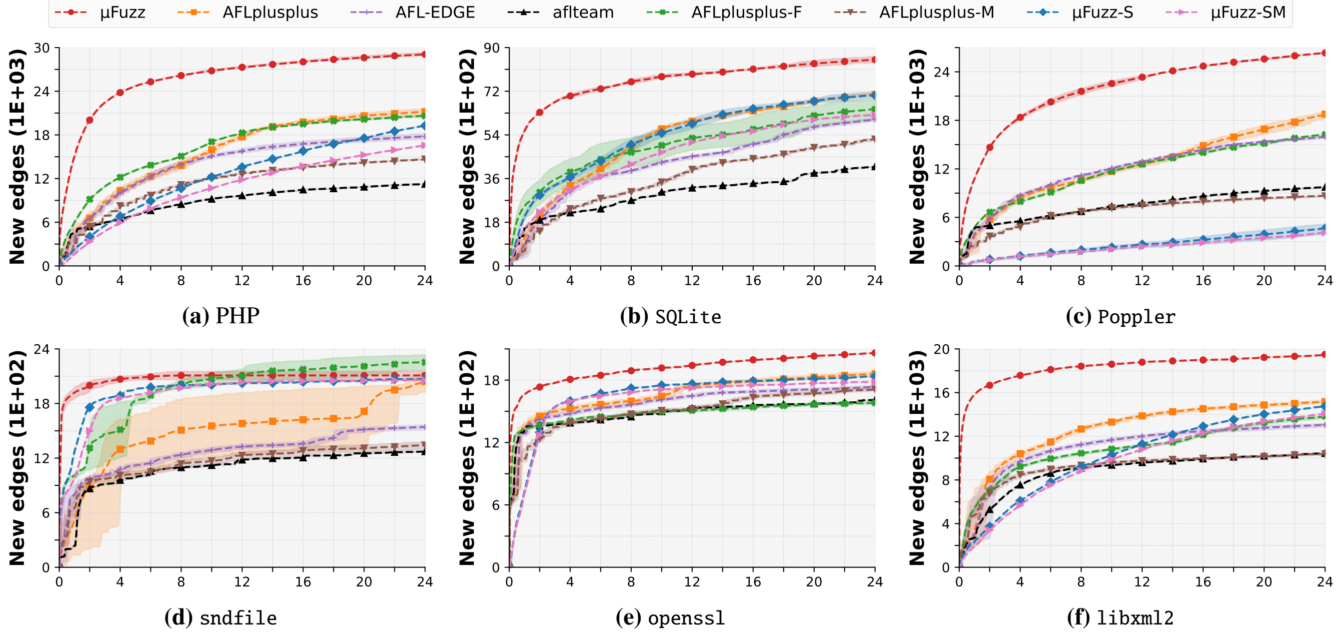


**Fig.5: Edge coverage found by evaluated fuzzers with 40 cores for 24h.** AFLplusplus-F is `AFLplusplus` with a synchronization interval of 30 seconds, an empirical value that can reach higher code coverage with lower overhead based on our testing. AFLplusplus-M is `AFLplusplus` with $\mu$FUZZ seed scheduling. $\mu$FUZZ-S is $\mu$FUZZ without state partitioning. $\mu$FUZZ-SM is $\mu$FUZZ without state partitioning and concurrency.

(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 input space is not well explored. An instance might find a bunch of interesting test cases, but cannot explore all of them timely. Under this situation, faster synchronization allows other instances to catch up with the progress and help explore the interesting test cases. A small corpus also allow the instances to synchronize with low overhead. However, as the code coverage is about to be saturated, there are not as frequent progress updates as in the beginning and AFLplusplus-F still synchronizes frequently. Every time an instance wants to synchronize with another's corpus, it has to walk through the directory to see whether there are any new test cases. Since the corpus has grown bigger, such operations become more expensive, so the fuzzing speed of AFLplusplus-F goes down, resulting in a slower increase in code coverage. If we further check the bug triggering time in Table 3, we can see that seven out of the 9 bugs found by AFLplusplus-F are within the first four hours. This is when faster synchronization is still beneficial for fuzzing.

Overall, $\mu$FUZZ outperforms the three compared parallel fuzzers `AFLplusplus`, `AFLEdge`, `AFLTeam`, and their variants in both bug detection and code coverage. The fuzzing effectiveness of $\mu$FUZZ comes from both its concurrent design and state partition.

## 5.3 Contribution of Microservice Architecture and State Partition

We compare $\mu$FUZZ with $\mu$FUZZ-S, which is $\mu$FUZZ with blocking synchronization but without state partition, to understand their contributions in parallel fuzzing. Since $\mu$FUZZ-S introduces blocking I/O, we further compare it with $\mu$FUZZ-SM, which is $\mu$FUZZ-S without concurrency and runs services synchronously. More specifically, every worker in $\mu$FUZZ-S maintains a copy of the global state. Whenever there are state updates, $\mu$FUZZ-S will propagate the updates to all its workers synchronously. The workers cannot handle new inputs until the state update is finished.

**Bug Detection.** $\mu$FUZZ successfully identifies 12 bugs in the targets in 24 hours, while both $\mu$FUZZ-S and $\mu$FUZZ-SM finds only 8. As shown in Table 3, $\mu$FUZZ finds all the bugs that are found by $\mu$FUZZ-S and $\mu$FUZZ-SM much faster, spending 46.6% and 53.7% less of the time respectively. Although $\mu$FUZZ-S and $\mu$FUZZ-SM found the same eight bugs, $\mu$FUZZ-S found five of them faster. We checked the fuzzing speed of these fuzzers to investigate the cause of the performance gap. We found that although $\mu$FUZZ-S has blocking synchronization, its fuzzing speed has no significant difference from that of $\mu$FUZZ, which has no blocking synchronization. This is because the concurrency allows $\mu$FUZZ-S to run other services during the state updates; thus its overall progress will not be blocked. However, the fuzzing speed of $\mu$FUZZ-SM is only 91% of $\mu$FUZZ-S on average. This means that in $\mu$FUZZ-S, when some services get blocked during synchronization, the other services cannot take over the CPU and

**Table 3: Bug Detection Results in 24 Hours.** We measure the bug detection capability in the number of identified bugs and their survival time. Targets are the top six programs with the most inserted bugs from the `Magma` benchmark. The bug ID is a unique identifier for the inserted bug in `Magma`. Results for `openssl` and `PHP` are excluded because no fuzzers find any inserted bug in these targets in 24 hours. We repeat the experiment for 5 times. For each bug and fuzzer, we use `Magma` to calculate its average *survival time* across the 5 runs, which indicates the time it takes for the fuzzer to find the bug. If the fuzzer cannot find the bug in 24 hours, we mark the survival time as ∞. The time highlighted in green means the corresponding fuzzer is the fastest to find the corresponding bug.

| Targets | Bug ID | μFuzz | AFLplusplus | AFLEdge | AFLTeam | AFLplusplus-F | AFLplusplus-M | μFuzz-S | μFuzz-SM |
|---|---|---|---|---|---|---|---|---|---|
| Poppler | PDF010 | 12h56m | 19h18m | 22h13m | **05h31m** | 13h13m | 18h19m | ∞ | ∞ |
| | PDF016 | **01m40s** | 16m40s | **01m40s** | 16m40s | **01m40s** | **01m40s** | 03h21m | 03h25m |
| | PDF021 | **05h22m** | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| sndfile | SND017 | 01h24m | 03h34m | 02h51m | 02h46m | 16m45s | **16m40s** | 02h46m | 01h25m |
| | SND020 | **01h24m** | 04h15m | 02h51m | 02h46m | 02h50m | 02h50m | 02h46m | 01h25m |
| libxml2 | XML002 | **12h57m** | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | XML003 | **02h46m** | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | XML009 | **16m40s** | 02h46m | 02h46m | 02h46m | 02h46m | 02h46m | 03h06m | 03h31m |
| | XML012 | **15h13m** | ∞ | ∞ | ∞ | 20h45m | ∞ | 20h07m | 22h38m |
| | XML017 | **01m40s** | 16m40s | **01m40s** | **01m40s** | **01m40s** | 16m40s | 02h48m | 02h46m |
| SQLite | SQL002 | **02h46m** | 10h34m | 07h28m | ∞ | 03h16m | 07h53m | 06h19m | 09h14m |
| | SQL018 | **02h46m** | 02h52m | 05h14m | ∞ | 03h40m | ∞ | 03h33m | 07h10m |
| **Total Bugs Found** | | **12** | **8** | **8** | **6** | **9** | **7** | **8** | **8** |

utilize the computation power. Therefore, we can have two conclusions: First, with concurrency, the fuzzer continues to make progress in the existence of blocking I/O and thus well utilizes the CPU power. Second, with state partition, μFuzz can find more bugs faster.

**Code Coverage.** As shown in Fig. 5, μFuzz finds 96% and 117% more edge than μFuzz-S and μFuzz-SM respectively in the six targets on average. From Fig. 5, we find that the coverage increase of μFuzz is much faster than μFuzz-S until the coverage is saturated. We checked the execution status and found that workers in the corpus management service of μFuzz-S tend to select duplicated test cases for mutation. This is because the workers have the exact same state as each other and use the same scheduling algorithm. Such duplication can slow down fuzzers' exploration. μFuzz-SM has the worst performance because it not only suffers from the aforementioned duplication but also slower fuzzing speed due to blocking I/O.

Overall, μFuzz outperforms μFuzz-S and μFuzz-SM in both bug detection and code coverage. Both the microservice architecture and state partition contribute to μFuzz's strengths. The concurrent design allows μFuzz to make progress in the existence of blocking I/O. State partition allows all the workers to work independently and still achieves an overall optimal result.

### 5.4 Identified New Bugs

Since we do not propose any new fuzzing strategy but help the existing strategies parallel more efficiently, we only ran some preliminary experiments to fuzz some programs from OSS-Fuzz for a short time. Yet we still find three new bugs in these well-tested programs, 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 three identified bugs include one logical error and two memory-corruption, of which one has been fixed and one acknowledged by the developers at the time of writing. We omit the details currently for anonymity. We are confident in μFuzz since it has better performance than existing fuzzers according to our evaluation.

## 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, which allows all threads to share the same memory space so that they can share data efficiently. μFuzz can easily 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), which is the state-of-the-art approach. For example, we can run μFuzz on different machines and connect their corpus management to synchronize the corpus and connect their feedback collection to synchronize the code coverage bitmap. More importantly, although state synchronization

over the network can cause slow I/O, μFUZZ will not be affected by this kind of I/O thanks to its concurrency design. If one service need to wait for the network communication to complete, the other services can still run and make individual progress. Another way is to utilize the microservice architecture to run different services on different machines and communicate over the network. We only need to add input caching to each service. In this way, each service can keep fetching the result from other services into the input cache and running the workers to consume the inputs concurrently. As long as we warm up each service by filling enough results in the input cache before fuzzing, each service can run without waiting for network I/O.

## 6.2 Support More Mutation Strategy

Since μFUZZ focuses on better parallelizing existing fuzzing strategies instead of inventing new ones, the current implementation of μFUZZ only supports the basic mutation strategy: `AFLplusplus`'s havoc mutation. The havoc mutation only performs bit-level or byte-level unstructured mutation. This limits μFUZZ's effectiveness on fuzzing targets that require structure-aware or semantic-aware inputs. To further improve μFUZZ's applicability, we plan to integrate μFUZZ with `LibAFL` [30], a recently open-sourced reusable fuzzing development kit that implements many reusable advanced fuzzing strategies. Thanks to the modularized design of μFUZZ, we can easily incorporate advanced strategies by modifying only the mutation service without changing the whole framework.

## 6.3 Support Collaborative Fuzzing

The current implementation of μFUZZ does not support collaborative fuzzing, which combines all kinds of different fuzzers to get a higher overall fuzzing performance. This is because μFUZZ now assumes that each service maintains one united global state and each worker inside the service maintains one partition of the state. However, different fuzzers can have different fuzzing states for the same functionality and μFUZZ cannot distinguish workers of different states inside a service. For example, grammar fuzzers might maintain a corpus in the form of abstract syntax trees (AST) instead of a binary stream. Suppose we combine a grammar-based mutation along with a bit-level mutation in μFUZZ. In that case, the input dispatcher of the test case generation service might wrongly dispatch an AST test case to a worker of bit-level mutation. We plan to try two ways to support collaborative fuzzing with μFUZZ. First, we can tag both the data of service communication and the workers. Moreover, we restrict the input dispatcher to only dispatch inputs to workers with matching tags. In this way, a service can maintain different types of workers and different states. Another way is that we can use μFUZZ as a base fuzzer for existing collaborative fuzzing approaches. For example, we want to combine 20 instances of a bitflip fuzzer and 10 instances of a grammar fuzzer. Then we can implement both fuzzers in μFUZZ, and then use them as the base fuzzers in the existing collaborative fuzzing framework such as ClusterFuzz [4].

# 7 Related Work

Existing works on scaling fuzzing performance can be mainly divided into two categories: vertical and horizontal scaling [10]. Vertical scaling refers to improving the performance of the internal components of a single-instance fuzzer, such as improving fuzzing strategy or fuzzing speed. Horizontal scaling refers to improving the performance of parallel fuzzing.

## 7.1 Fuzzing Strategy Improvement

Improving the fuzzing strategy focuses on enhancing the internal components of a fuzzer, which can include 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 a model that describes the format of the inputs to generate structural inputs that can reach deeper logic of the software. SQLSmith [1] uses the SQL grammar and database schemes to generate more valid queries. MoWF [57] leverages the file format information to fuzz the deeper program code beyond the parser. Mutation-based fuzzing differs from generation-based fuzzing in that it performs mutation on existing test cases to generate new ones. In this way, the fuzzer can use various feedback information collected from the execution phase to guide its test case generation. AFL [79] uses edge coverage to model program states to guide its mutation, which is highly effective. The mutation strategy, feedback quality, and seed scheduling algorithm can all influence the performance of mutation-based fuzzers. 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.

$\mu$FUZZ does not improve existing fuzzing strategies. Instead, it focuses on better parallelizing these strategies. With its microservice design, existing fuzzing strategies can be easily integrated into the corresponding services and achieve a better result in parallel fuzzing.

## 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. UnTracer [52] proposes coverage-guided tracing to trace block coverage only when new ones are discovered. Nagy et al. [54] further extend the idea of coverage-guided tracing to support edge coverage recording. 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 tailed 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 $\mu$FUZZ 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.

Instead of building on top of existing single-instance serial fuzzers, $\mu$FUZZ redesigns parallel fuzzing with microservice architecture. It mitigates the problem caused by blocking I/O and periodic state synchronization.

# 8 Conclusion

We present $\mu$FUZZ, a parallel fuzzing framework in microservice architecture that supports concurrency to improve CPU utilization in the existence of blocking I/O and avoids state partition to fuzz with a globally optimal strategy. Our evaluation shows $\mu$FUZZ is more effective in parallel fuzzing than existing fuzzers with 57% improvement in code coverage and 67% improvement in bug detection on average in 24 hours. Besides, $\mu$FUZZ finds three new bugs in well-tested real-world programs.

## 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] Scalability. https://en.wikipedia.org/wiki/Scalability, 2022.

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

[12] strace is a diagnostic, debugging and instructional userspace utility for linux. https://github.com/strace/strace, 2022.

[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 Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. 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. 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.

[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.