Monarch: A Fuzzing Framework for Distributed File Systems

Tao Lyu *Liyi Zhang*[*] *Zhiyao Feng* *Yueyang Pan* *Yujie Ren*
Meng Xu[*] *Mathias Payer* *Sanidhya Kashyap*

EPFL [*]University of Waterloo

Abstract

Distributed file systems (DFSes) are prone to bugs. Although numerous bug-finding techniques have been applied to DFSes, static analysis does not scale well with the sheer complexity of DFS codebases while dynamic methods (e.g., regression testing) are limited by the quality of test cases. Although both can be improved by pouring in manual effort, they are less practical when facing a diverse set of real-world DFSes. Fuzzing, on the other hand, has shown great success in local systems. However, several problems exist if we apply existing fuzzers to DFSes as they 1) cannot test multiple components of DFSes holistically; 2) miss the critical testing aspects of DFSes (e.g., distributed faults); 3) have not yet explored the practical state representations as fuzzing feedback; and 4) lack checkers for asserting semantic bugs unique to DFSes.

In this paper, we introduce Monarch, a multi-node fuzzing framework to test all POSIX-compliant DFSes under one umbrella. Monarch pioneers push-button fuzzing for DFSes with a new set of building blocks to the fuzzing toolbox: 1) A multi-node fuzzing architecture for testing diverse DFSes from a holistic perspective; 2) A two-step mutator for testing DFSes with syscalls and faults; 3) Practical execution state representations with a unified coverage collection scheme across execution contexts; 4) A new DFS semantic checker SymSC. We applied Monarch to six DFSes and uncovered a total of 48 bugs, including a bug whose existence can be traced back to the initial release of the DFS.

1 Introduction

Distributed file systems (DFSes) are the backbone of modern computing infrastructure. In particular, various data-intensive applications, such as AI model training [61] and high performance computing [1, 56] rely on them for reliably storing data. However, providing reliability for DFSes is challenging. First, DFSes are often developed in low-level programming languages (e.g., C/C++), which are susceptible to memory bugs [59]. Second, the nature of the distributed environment adds another level of complexity in the form of concurrency and fault tolerance, often leading to semantic bugs. All these bugs in DFSes pose significant threats, such as information leakage [40], remote code execution [53], privilege escalation [41], and data loss or corruption [58]. Therefore, to improve their reliability and security, finding and fixing bugs in DFSes becomes critical.

Various bug-finding techniques, such as model checking [36, 66], formal verification [24, 63], and dynamic testing [6, 14], have been applied to DFSes with different levels of success. However, these techniques suffer from a common pitfall – their effectiveness strongly correlates with the amount of manual effort. Formal verification and model checking, for instance, require significant expertise and time to construct an abstraction model to avoid state explosions; while dynamic approaches depend heavily on the quality of test suites, of which a manual construction would often miss bug-revealing corner cases.

On the other hand, fuzzing is one of the most popular testing techniques, especially for its ability of automated exploration of execution state space and rare false alarms [9]. For instance, several works have found over hundreds of bugs in file systems [16, 34, 64, 65, 68], indicating the practicality of fuzzing. However, file systems fuzzers specifically target *local* file systems (LFSes), which run on a single node. This paper focuses on a seemingly yet more challenging question: *how can we adapt the effectiveness of local file system fuzzing to the complexity of DFS?* While directly applying LFS fuzzers to DFSes might seem intuitive, such an approach is inadequate for several reasons.

First, existing fuzzing architectures are limited to feeding inputs to, gathering execution states from, and monitoring bug exposure within a single execution context (e.g., the kernel space in one node). However, DFSes comprise of components that run across computing nodes and contexts. Therefore, DFS fuzzing requires a holistic infrastructure to set up DFSes and feed inputs to them across multiple nodes, monitor the status of each node, and assert bugs by aggregated state information. Additionally, we need “practical” representations of the entire DFS execution state for fuzzing feedback – ones that are easy to instrument, low-overhead to capture at runtime, and adequate as a state-space approximation. Furthermore, LFS fuzzers cannot fully explore the input space of DFSes. Contrary to the conventional (e.g., Syzkaller) notion that the input for LFS fuzzing is a sequence of syscalls, DFSes are designed to take concurrent syscall sequences and to tolerate random distributed faults. To effectively explore the state space of DFSes, a systematic mutator that can generate test cases with both syscalls and faults is essential. Finally, a DFS fuzzer needs bug checkers that can handle a variety of inputs (e.g., syscalls and faults) and subsequently deduce whether the concrete execution is specification (e.g., POSIX) compliant. With such bug checkers, the fuzzer can find not only memory errors but also semantic bugs that are traditionally only uncovered via hand-tuned model checking.

To address the aforementioned issues, we propose **Monarch**: a general fuzzing framework for POSIX-compliant DFSes. As a highlight, Monarch incorporates the following key designs:

VM-based multi-node fuzzing architecture [§3.1] Monarch proposes the VM-based controller-worker model for multi-node DFSes fuzzing. The controller tests DFS instances in workers with each worker comprising multiple VMs dedicated to a single DFS instance, together with the executor, coverage, and checker agent in each VM.

Test case generation [§3.2] Monarch employs a two-step mutator for testing syscalls in non-fault mode and both syscalls and faults in fault modes flexibly.

Execution state representation [§3.3, §3.4] Monarch adopts distinct and practical execution state representations, uniformly collected across execution contexts, for the fault and non-fault testing modes. Accordingly, it utilizes a test case reduction scheme to improve the fuzzing efficiency.

Bug checker [§3.5] Monarch detects memory bugs and various types of semantic bugs in DFSes using memory bug detectors and a home-grown semantic checker SymSC.

We evaluate Monarch on six DFSes and found 48 bugs, including 26 memory bugs triggered by syscalls, 14 memory bugs triggered by syscalls and faults and eight semantic bugs.

Summary. This paper makes the following contributions:

- **Multi-node fuzzing architecture**: To the best of our knowledge, Monarch is the first multi-node DFS fuzzing framework, includes all the basic building blocks for fuzzing various POSIX-compliant DFSes.
- **Semantic checker for DFSes**: We categorize semantic bugs in DFSes into different types, further, design a new semantic checker to uncover all of them.
- **Impact**: Monarch has found 48 bugs in six popular DFSes, including 40 memory bugs and eight semantic bugs.

Monarch is publically available at `https://github.com/rs3lab/Monarch`.

DFS	Role	Execute Context		Backend	FT
		Server	Client		
GlusterFS	SR	U	U	LFS	✓
BeeGFS	MR	U	K	LFS	✓
CephFS	MR	U	U, K	Own [2]	✓
OrangeFS	SR	U	U, K	LFS	×
NFS	SR	K	K	LFS	×
Lustre	MR	K	K	LFS	✓

Table 1: Different designs of DFSes from various perspectives. **SR**: single-role, **MR**: multi-role. Servers and clients are running in the execution context either in user space (**U**), in kernel space (**K**) or in both (**U,K**). For storage backends, servers can use either unmodified local file systems (**LFS**) or their own storage backend (**Bluestore**). **FT** represent the support for fault tolerance in a DFS.

2 Background and Motivation

In this section, we briefly introduce diverse architectures of popular DFSes, summarize common bug types in DFSes through a sample of bugs found by Monarch, and highlight the gaps between existing fuzzers and the ideal DFS fuzzer.

2.1 Architectural Diversity in DFSes

A DFS consists of clients and servers in varied configurations. Clients send requests to servers in the form of file system-specific operations, while servers respond to clients and communicate with each other to manage data and metadata. We categorize architectural diversity in DFSes in Table 1.

First, based on the server functionality, we classify a DFS into (1) single-role (SR) architecture, where a server node manages both data and metadata [7, 12, 30]; and (2) multi-role (MR) architecture, where each server node is only responsible for either data or metadata, but rarely both [20, 44, 57, 62]. Second, DFS components (*e.g.*, a server or a client) can execute in either user context or kernel context or both. For example, `GlusterFS` is a userspace file system, while `NFS` and `Lustre` are kernel-based ones. `CephFS` implements the server logic in userspace, while providing both in-kernel and FUSE-based userspace clients [62]. Another category is storage backend, a DFS either use its own backend [62] or an existing LFS [10, 25]. Finally, in terms of the fail-safe behaviors, some DFSes provide fault tolerance, a critical requirement for avoiding data loss and improving availability. For example, some DFSes [12, 20, 44, 62] employ replication mechanisms in case any nodes crash or are partitioned from the cluster.

2.2 Bugs in DFSes

There are two main categories of bugs in DFSes. First, many DFSes are developed with memory-unsafe languages (e.g., C/C++) for performance reasons, making them prone to *memory bugs*. Second, similar to LFSes, DFSes expose interfaces (*i.e.*, syscalls) with specified semantics (*e.g.*, POSIX [29]) to users. Incorrect implementation of such interfaces can violate the defined semantics, leading to *semantic bugs*. Moreover, the distributed nature of DFS introduces several complex logic, such as cross-node state sharing, concurrent access, and fault tolerance. Thus, triggering and detecting both types of bugs in DFSes pose non-trivial challenges compared with those in LFSes. We now detail them with examples below.

Memory bugs. Memory bugs (*e.g.*, buffer overflows, use-after-free) often lead to severe consequences, such as remote code execution or denial-of-service [59]. Figure 1 illustrates a use-after-free bug in a userspace client of `GlusterFS` that Monarch finds by crashing or disconnecting one of the two servers before executing `removexattr` [50]. Although LFS fuzzers [16, 34, 65] can find and detect memory bugs [21, 33, 55], they fail to find this bug because *(1) they are not designed for testing multiple cross-node and userspace file systems, (2) they lack fault injection mechanisms, like node crashes or network partitions, to trigger such bugs.*

```
mkdir("./A", 0777);
setxattr("./A", "user.attr", "val", 3, 0);
// A server crashes or partitioned from the client.
removexattr("./A", "user.attr");
// Client crashes during executing removexattr.
```

Figure 1: A memory bug triggered by syscalls and a fault.

Semantic bugs. The semantics of a DFS are encoded in its exposed interfaces which, on POSIX systems, are typically file system-related syscalls. Each syscall is akin to a contract between a DFS and a user in terms of how the internal file system state should transit, *i.e.*, when modeling a DFS as a state machine, a syscall can transit its internal state from s_0 to a set of new states $\{S\}$. It is a semantic bug if the DFS lands on state $s_1 \notin \{S\}$ after the syscall. We now categorize such bugs into four types.

(1) Semantic violations on in-memory states (SVM). Every file system maintains in-memory states for reducing storage media access latency. POSIX specifies the in-memory state transition for each syscall [52], which must be completed before a syscall returns.[1] File systems violating the in-memory specifications lead to bugs, which we refer to as semantic violations on in-memory states (SVM). Figure 2 shows such a violation found by Monarch [47]. If we execute an `open` with `O_DIRECTORY|O_CREAT` flag in `GlusterFS` backed by LFSes, POSIX states that the behavior of this flag is unspecified [15], resulting in non-deterministic return values on LFSes. However, `GlusterFS` assumes this operation to be deterministic. This leads to an incorrect `GlusterFS` in-memory state, causing two consecutive `stat` to return different results.

```
open("./file1", O_CREAT|O_DIRECTORY|..., 0330)
stat("./file1", stat_buf) // Success
stat("./file1", stat_buf) // ENODATA (No data available)
```

Figure 2: An **SVM** bug found by Monarch that two consecutive `stat` see different results.

[1]We do not cover asynchronous calls as they are beyond POSIX scope.

(2) Semantic violations on persistent states (SVP). File systems persist data by flushing the in-memory states to the disk either periodically or on user-issued persistence syscalls (e.g., `fsync`, `fdatasync`, `sync`). POSIX specifies that data and metadata flushed by persistence syscalls need to stay in a consistent state after a crash and recovery, also known as crash consistency. Notably, in the context of DFSes, after a complete (i.e., all-nodes) or partial crash and recovery, a POSIX-compliant DFS must be crash-consistent though it may not be fault-tolerant. Violations of such specifications are semantic violations on persistent states (SVP) or crash inconsistency bugs, which lead to dire consequences (*e.g.*, data loss), and further result in misbehaviors of applications running on top of the DFS.

Figure 3 illustrates a violation of the crash consistency property found by Monarch [48]. In this case, we explicitly persist a directory (`B`) onto disk using `fsync` right before crashing all servers. However, `B` still gets lost after recovery from the crash. This bug originates from a flawed implementation in `GlusterFS` servers, where servers respond a success to the client regarding the `fsync` on a directory without actually flushing the directory onto the disk. While tools detecting crash consistency bugs are available for LFSes [34, 43, 67], none of them can be directly applied to DFSes *due to the incapability of controlling and testing cross-node DFSes.*

```
// Create a directory named B under the mounted client.
mkdir("A/B", 0777);
// Persist the mount directory and B.
int parent_fd = open("A/", O_RDONLY|O_NONBLOCK);
fsync(parent_fd);
int dir_fd = open("A/B", O_RDONLY|O_NONBLOCK);
fsync(dir_fd);
// Crash servers immediately.
```

Figure 3: An **SVP** bug detected by Monarch. The directory is lost after a crash and recovery, even after flushing using `fsync`.

(3) Semantic violations under fault states (SVF). LFSes follow the fail-stop model. Hence, any fault in a LFS can corrupt the file system, sometimes leading to entire system crashes. However, in DFSes, syscalls either proceed according to POSIX as if no faults are happening, or return errors, depending on the fault state and certain criteria specific to each DFS (defined as fault models in §3.5). Thus, the semantics of syscalls under faults are changed and redefined together by POSIX and DFS fault models. We call violations to such semantics as semantic violations under fault states (**SVF**).

Figure 4 presents an SVF with a `GlusterFS` instance consisting of 3 replica servers [51]. A directory `"A"` is created and its extended attribute is set to `"user.key:val"`. Unfortunately, one of the replica servers `srv1` crashes soon after that. According to the fault model in `GlusterFS`, a syscall proceeds conforming to POSIX as long as over 51% of replica servers are online [13]. Therefore, the subsequent `removexattr` removes the attribute successfully. However, later, when `srv1` is brought back, `GlusterFS` does not synchronize it with other nodes to remove the attribute stored on it. That could leave a possibility that a `getxattr` on `"A"` reads the stale attribute from `srv1`, violating the semantics specified by both POSIX and the fault model. *Being unaware of fault models, existing LFS fuzzer cannot detect such bugs.*

```
mkdir("./A", 010);
setxattr("./A", "user.key", "val", 3, 0);
// Replica server 1 crashed
removexattr("./A", "user.key");        // Success
// Replica server 1 comes back
getxattr("./A", "user.key", value, 3); // Success
```

Figure 4: A **SVF** manifests under partial faults.

Additionally, to check syscall semantics under different fault states, we have to inject faults to transit among fault states, which motivates us to add faults as an input space.

(4) Semantic violations under concurrent executions (SVC). DFSes, as shared resources, must support concurrent operations from multiple clients. Thus, in contrast to the per-syscall state transitions mentioned above, in concurrent execution, the file system in-memory and persistent state transits to the next one through a group of concurrent syscalls. Unfortunately, POSIX does not specify the behavior of concurrent file system operations. Thus, LFS fuzzers, designed for sequential testing, fail to detect semantic violations under concurrent executions (SVC).

Figure 5 is an example of SVC in `CephFS` reported by Monarch [49]. The bug manifests itself when two clients execute syscalls concurrently, and there is such an interleaving below. When the metadata server receives a `chmod` request on `inode A` from `client 1`, it creates a projected inode `inode A′`, applies a per-inode exclusive lock `authlock` to and modifies permissions on `inode A′`, queues `inode A′` for journaling, and promptly responds to `client 1`. Subsequently, both clients send stat requests simultaneously. As `client 1` retains the lock, it can access `inode A′` with updated metadata. However, `client 2` retrieves outdated metadata of `inode A` after failing to get the `authlock` instead of waiting for the `authlock`. Thus, two clients see inconsistent metadata of file `A` at the same time.

```
// client 1                          // client 2
open("A", O_CREAT|..., 012);         open("A", O_RDWR|..., 000);
chmod("A", 000);
stat("A");                           stat("A");
// mode: 32768                       // mode: 32776
```

Figure 5: A **SVC** found by Monarch in `CephFS`. Two `stats` read out different file modes after concurrent executions.

`CephFS` complies with the strong consistency guarantee in POSIX specification [5]. Hence, two clients should retrieve the same file states in the above case. In other words, the latest state of `"A"` updated by `chmod` should be returned to *Client 2* when executing `stat`. Because `stat` in *Client 2* is issued after `chmod`. The maintainer has confirmed this bug, and its patch is under internal review [49]. This inconsistency among clients can lead to confusion and incorrect workflows in applications. Similarly, existing tools cannot uncover this bug, *because they are unable to check the semantics of concurrent syscalls.*

2.3 Fuzzing for DFS: The Missing Pieces

Fuzzing, or fuzz testing, is a dynamic program analysis approach that has gained traction in finding bugs in large and complex real-world software. Generally, a fuzzer continuously mutates existing test cases (*i.e.*, seeds), which consist of **inputs** to the testing program (*e.g.*, syscalls for file systems), using predefined rules to generate new test cases. Further, new test cases are fed to the testing programs (*e.g.*, LFSes) for executions, during or after which, **checkers** assert bugs according to the runtime information. Intuitively, with an unknown distribution of bugs, the more execution states explored, the more likely a fuzzer can find a bug. *Coverage-guided fuzzers* typically adopt an evolutionary process, which saves test cases hitting new execution states as seeds for further mutations on them. Specifically, **execution state representations** are expressed as the branch coverage from single-node and single-context testing programs. Furthermore, to improve mutation and execution efficiency, seeds are **reduced** into disjoint and independent pieces, such as syscall sequences that do not impact the executions of each other, yet collectively produce the same execution states.

Existing fuzzers are specifically designed for userspace program [17], kernel and LFSes [16, 34, 64, 65], and network protocols [8, 45, 54]. Although DFS fuzzing shares the same workflow described above with them, most building blocks still need a complete re-design, as detailed below.

Missing piece 1: Multi-node fuzzing architecture. LFS fuzzers target file systems on a single node. They fail to set up a DFS and control its concurrent executions across multiple nodes, as well as monitor the status of nodes for asserting bugs. Therefore, it necessitates a multi-node fuzzing architecture to test all DFS components in a holistic view.

Missing piece 2: Distributed faults as a fuzzing input space. As the SVM example in Figure 1 and SVF examples in Figure 4 illustrate, distributed faults, 1) can trigger incorrect memory bugs in the fault-handling code, 2) enforce exploring different fault states for exposing semantic bugs. Unfortunately, none of the existing fuzzers have systematically explored how to mutate faults and syscalls together.

Missing piece 3: Sufficient and low-overhead representation of DFS execution states. Programs (*e.g.*, LFSes) that existing fuzzers target are executed entirely in a single address space (i.e., one process in userspace [34, 65] or the kernel space [16]). Their execution states can be represented by the entire branch coverage in their own address space. However, DFSes have multiple cross-node components and coverage instrumentation on each component brings overhead. Therefore, a research question comes into the picture: *What would be the "practical" representations of cross-node and cross-context execution states?* Essentially, we need representations that are low in runtime overhead and sufficient for approximating the execution state space. Additionally, DFS components run either in the user or kernel context. A

DFS fuzzer needs to track code coverage uniformly across both contexts if multi-component coverages are required.

Missing piece 4: Semantic checker for DFSes. Distributed faults and concurrent executions across clients bring semantic bugs unique to DFSes, such as the SVF in Figure 4 and SVC in Figure 5. Detecting these types of bugs requires a new semantic checker, as none currently exists.

3 Design

We propose Monarch, a fuzzing framework for POSIX-compliant DFSes, to replicate the success of LFS fuzzing in a distributed setting. We now discuss how we fill the aforementioned gaps with Monarch for fuzzing DFSes.

3.1 Monarch Architecture

Monarch adopts a *VM-based multi-node fuzzing architecture* to cover various DFS architectures (refer to Table 1). Monarch uses VMs rather than containers or library OSes, as they are incompatible with DFSes. For instance, some DFSes consist of both kernel and userspace modules, which rules out container-based nodes. Moreover, most DFSes are multi-processes, as they use a set of daemons to interact with other nodes [12, 62]; while the current multi-processing support in library OSes are not mature enough.

Figure 6 illustrates the architecture of Monarch, which is based on the controller-workers model. The controller is a centralized component consisting of (1) a mutator for generating test cases and distributing them to workers; (2) a tracker that receives and merges code coverage from workers and performs test case reduction; and (3) a checker that applies the *distributed bug-checking logic*, which asserts bugs by analyzing the collected per-node states from each checker agent. Each worker sets up a complete DFS instance that spans across multiple nodes for executions. Each node contains one DFS component (e.g., either a server or a client), an executor for executing syscalls and launching faults included in test cases, a coverage agent (coverager) for collecting code coverage on this node, and a checker agent for monitoring and extracting node-specific DFS states, such as metadata associated with the DFS, syscall execution timestamps, or any process crashes. Notably, executors (1) set up servers and mount the DFS for clients with predefined configuration scripts, (2) extract a node-specific slice from the multi-node test case received by workers, (3) run the tasks dictated in the test case slice (e.g., faults on servers or syscalls on clients), and (4) synchronize the execution and tear down nodes.

3.2 Mutating Inputs in a DFS

We segregate inputs to a DFS that impact its states into two types. The first type is the **explicit** state modification by either issuing syscalls or remounting a disk with altered

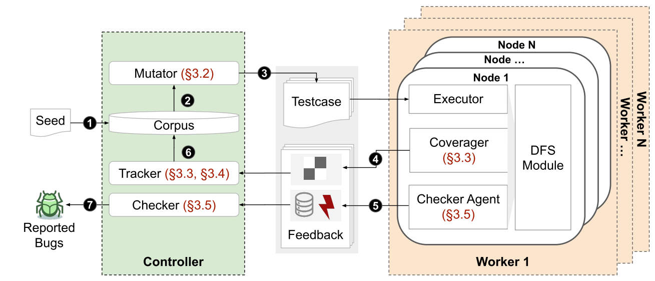

Figure 6: Monarch architecture. When starting a fuzzing instance, user has the option to provide test cases for initializing the *Corpus* (❶). The *Mutator* (❷) mutates a seed selected from the corpus or generate test cases according to pre-defined syscall dependency if the corpus is empty. The mutated or generated test cases are distributed to *Executors* (❸) for testing. During the execution, *Coverage* (❹) collects coverage and finally set back to the *Tracker* (❻) for selecting seeds. Moreover, *Checker Agents* (❺) forward DFSes states or crashes to the *Checker* (❼) for reporting bugs to users.

states. Although both have been adopted as testing inputs in LFS fuzzers [34, 65], syscalls are more critical in DFS because most DFSes (except `CephFS`) rely on LFS layer as a storage backend, which directly interacts with the disk and has already handled any on-disk states. Thus, Monarch only uses syscalls. The second is the one that **implicitly** modifies DFS states, such as crashes and network partitions. Overall, Monarch broadens the fuzzing scope to include both explicit (*i.e.*, syscalls) and implicit inputs (*i.e.*, faults).

Monarch generates a test case consisting of sequences of calls, with each sequence designated for one node. In particular, each test case includes a set of: 1) file operation syscalls and 2) pseudo syscalls for fault injection (Table 2). Figure 7 shows a sample test case. Moreover, Monarch combines both inputs with a two-step mutator, which works as follows: it first prioritizes syscall mutations and then (optionally) fault mutations. This approach leads to two fuzzing modes: a *non-fault* mode that tests a DFS with syscall mutation only, and the *fault* mode that uses both input types.

3.2.1 Syscall mutation

File system syscalls, as user interfaces to DFSes, modify the DFS in-memory and persistent state of a file with valid arguments. Moreover, syscalls have correlation and data dependencies. For example, syscall `write(fd, ...)` depends on the returned file descriptor `fd` of `open("A", ...)`, meanwhile syscalls `open("A", ...)` and `chmod("A", 0)` are correlated because they operate on the same file and affect the execution of each other when executing concurrently.

Similar to existing approaches [22], Monarch also adopts a predefined syscall templates for generating valid syscall sequences. As a result, templates ensure that the generated syscall sequence adhere to the dependency and argument constraints of syscalls specified in the template. Furthermore, if a test case T discovers new execution states, Monarch

Pseudo syscalls	Description
`fault_start_barrier_c` `fault_stop_barrier_c`	A client notifies a server for starting or recovering from a fault. They return when the server has already started or recovered from a fault.
`fault_start_barrier_s` `fault_stop_barrier_s`	A server starts or recovers from a fault, either a network partition or node crash, when notified by all clients.

Table 2: The list of proposed pseudo syscalls in Monarch.

performs one of the following mutations on each syscall sequence T_i of T: (1) insert a new syscall or remove a syscall from T_i; (2) mutate arguments of a syscall in T_i; (3) select another slice T_j from the corpus and splice its partial call sequence T'_j at a random point of T_i.

Unlike LFSes, concurrent interaction with multiple clients is common in DFSes, which emphasizes the importance of generating concurrent syscalls from clients. Rather than constructing concurrent test cases from a fine-grained perspective (*e.g.*, memory accesses) done by prior works [31, 64], we choose a coarse-grained approach from the semantic perspective, which is simple and faster for test case generation. It utilizes the syscall correlation in DFSes to generate concurrent syscalls. As a result, Monarch retains the above generation and mutation strategies while improving the possibility that concurrent syscalls share the same file arguments.

3.2.2 Fault mutation

Testing the fault tolerance logic is critical to ensure the robustness of a DFS. As mentioned before (§2.2), the fault-tolerance logic is prone to both memory bugs (Figure 1) and semantic bugs (SVF in Figure 4). However, merely using the syscall mutation strategy is insufficient to stress the fault tolerance code. Hence, we design fault injection capability to alter and explore fault states. We now specify the types of injected faults and their granularity, and the algorithm.

Fault types. Both memory (Figure 1) and semantic bug (Figure 4) show that *node crashes* and *network partitions* are the two primary fault types contributing to bug-finding in DFSes. Our insight aligns with the distributed system bug characteristics [37]. Thus, Monarch only focuses on *network partitions* and *node crashes* as the fault injection types. Network partition refers to the disconnection between distributed nodes. It can either be a complete partition that divides nodes into completely disconnected groups, or a partial one, which separates nodes into indirectly connected groups. We introduce network partitions between servers and clients and among servers themselves. However, we do not introduce them among clients because they do not communicate. For *node crashes*, we only crash server nodes, which can have a significant impact on other nodes as they interact with each other to provide services to client nodes.

Fault granularity. Faults can occur at any point during the life cycle of a DFS. Thus, an ideal fault injection strategy

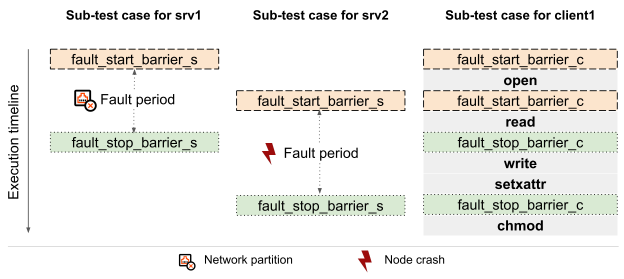

Figure 7: A test case of Monarch consists of several sequences of calls, including file operation syscalls and pseudo syscalls (for fault injections) shown in Table 2. Each call sequence is distributed to one node for execution. On clients, the call sequence includes POSIX file operation syscalls and pseudo syscalls. On servers, it only comprises pseudo syscalls as file operation syscalls are only executed at clients. For example, `fault_start_barrier_s` initiates faults, once it receives notifications from `fault_start_barrier_c`, and ensures that faults are launched before `fault_start_barrier_c` returns. Likewise, the `fault_stop_barrier_s` recover DFSes from faults when it receives notifications from `fault_stop_barrier_c`.

would be to simulate faults at the network packet boundary, *i.e.*, disconnect a node before and after sending/receiving a packet. However, it introduces a huge search space to inject faults and a much higher performance overhead as the executor needs to pause on every network packet to decide whether to fault. An alternative approach which Monarch adopts, is to *fault at the boundary of syscalls*, *i.e.*, we execute a syscall entirely inside or outside faults but not partially.

Enabling fault injection at the syscall boundary requires synchronizing faults with syscall execution. We achieve this by introducing pseudo-syscalls that act as a barrier before and after faults for clients and servers (see Table 2). Figure 7 shows a sample test case, in which two pseudo calls (`fault_start_barrier_c` and `fault_stop_barrier_c`) at the client synchronize with another two calls (`fault_start_barrier_s`, `fault_stop_barrier_s`) at servers to start and recover from faults. Finally, considering the high time cost of launching and recovering from faults, we limit at most one fault in each server for each test case.

Fault mutation strategy. Similar to the syscall mutation in §3.2.1, fault mutation is based on the feedback-driven test case generation principle for fault-state exploration. This method requires less manual effort compared to existing works [3, 11, 23, 38, 39, 60], which typically inject faults at limited points defined by users or heuristics, such as before or after accessing critical variables. Specifically, we 1) inject faults to a test case solely including syscalls, and 2) mutate the fault positions in a test case to generate new test cases if it hits new execution states during execution. We now present our simple yet effective algorithm detailing both scenarios.

For test cases without any injected faults, we first randomly choose a set of servers, and inject a fault into each syscall slice executed on these servers. Figure 7 shows the created slice for injecting faults (`fault_start_barrier_s` and `fault_stop_barrier_s`) in *srv1* and *srv2*. Later, we iterate over each syscall for every client to synchronize it with the injected fault, thereby generating a set of new test cases. This corresponds to adding `fault_start_barrier_c` and `fault_stop_barrier_c` in the slice executing on *client1*. Having only one syscall within the fault period is insufficient to trigger bugs. We overcome this limitation by including a random number of syscalls within the fault period that gets synchronized using the client-specific pseudo syscalls. If a test case yields new code coverage with injected faults, it indicates either the faults themselves trigger uncovered fault tolerance or recovery logic (representing a new fault state), or the syscalls executed during the faults experience new execution states. Therefore, we synchronize these faults with other syscalls in the test case to probe new execution states. This would be the process of adjusting the position of pseudo syscalls in the slices executing on the client.

3.3 DFS Execution State Representation

Similar to existing fuzzers, MONARCH adopts branch coverage as the unit of execution state representation. As raised in §2.3, a key question is to find a practical representation of DFS execution states. Given that bugs are scattered in any component of a DFS, an ideal way is to represent the execution state with coverages of all components. This can capture any coverage changes, thereby maximizing the likelihood of bug detection. However, it might not always be the most practical approach in practice. After experimenting with three different execution state representations by comparing their achieved coverage (see §5.2 for results), we summarize two findings and utilized them for deciding the representations in MONARCH.

Finding 1*: In non-fault mode, the collection of client coverage is a sufficient approximation to represent the execution states of a DFS.* The intuitions behind this are three-fold: 1) Code pieces for server and client components might have a one-to-one mapping (i.e., a code execution *E1* in the client corresponds to an execution *E2* in the server and vice versa). 2) The client might be dominant in DFSes even for DFSes that are considered client-light (i.e., the client only wraps syscalls and bridges the request to servers). For example, the `NFS` client has a nearly 2x code size than its server counterpart. 3) Servers and clients might share some codebases (e.g., Remote Procedure Call (RPC) libraries). These shared code can be considered as "explored" from a fuzzing perspective if they are executed from either side. For MONARCH, one branch in the code is always represented with a fixed ID in any environment. Thus, client coverage can always represent the coverage of shared code even if there is no server coverage.

Finding 2*: In fault mode, it is necessary to collect coverage from both servers and clients to represent the DFS execution states.* That is simply because the fault tolerance modules mostly run on servers that client coverage cannot represent.

Granularity of branch coverage collection. Test cases hitting new execution states (new branch coverage) are saved as seeds for future mutations, such as adding new syscalls. However, this can result in extremely large test cases, which, however, are of little practical value especially for 1) mutation efficacy, 2) execution efficiency and 3) debug-ability, which are detailed in §3.4. As a result, most LFS fuzzers (e.g., Syzkaller) track not only aggregated coverage of the entire test case, but also per-syscall coverage to enable dynamic test case reduction (detailed in §3.4).

Tracking per-syscall coverage is simpler in LFSes, where a single kernel thread handles each syscall, except in cases where background kernel threads are involved. In contrast, in most DFSes, each syscall involves a client thread and multiple server threads. Thus, tracking per-syscall coverage on the client side is still achievable. However, it becomes challenging on servers, as servers handle RPC requests and are typically not aware of syscalls that issue these RPC requests. Additionally, RPCs requested from clients might subsequently trigger new RPCs among servers, complicating the tracking. While it is feasible to instrument both clients and servers heavily for linking these RPC requests, it results in extra overhead and further reduces fuzzing efficiency. Thus, MONARCH tracks aggregated coverage per each test case at servers.

To summarize, MONARCH tracks per-syscall coverage at clients, while per the entire test case coverage at servers.

Unified coverage collection. As summarized in §2.1, DFSes consist of cross-context components such as in-kernel client and userspace server in `CephFS`. While Kcov [32] can track code coverage for in-kernel components (both per syscall and per test case), there is no such coverage collection scheme for userspace components. Therefore, we propose UCOV, for collecting both per syscall and per test case coverage in userspace DFS components.

(1) Ucov for DFS clients. Generally, DFSes design their userspace clients using FUSE—a kernel module under the Virtual File System (VFS) layer that forwards file system syscalls to a userspace client via device `/dev/fuse` and later receives the execution result through that device as well. We observe that the data structure `fuse_in_header` used between FUSE and the userspace program has an ID representing the thread issuing the syscall. This allows Ucov to collect per-syscall coverage at userspace clients by tracking this ID.

(2) Ucov for DFS servers. For DFS servers executing in userspace, Ucov builds a memory pool for DFS threads to retrieve and release coverage recording memory dynamically. To make a newly spawned thread aware of the coverage tracking memory, we instrument a thread-local pointer representing the coverage recording memory, which is default to `null` in every threads. When appending basic block IDs to the memory, the instrumentation code retrieves a new chunk of memory from the pool if the pointer is `null`. When the thread exits, the retrieved memory is released by a callback previously set using `pthread_key_create()`.

3.4 Dynamic Test Case Reduction

Both fuzzers and developers (when triaging bugs) prefer "small" (less syscalls) test cases. This is beneficial for: (1) Debug-ability: triaging a bug in hundreds of syscalls is like finding a needle in a haystack. (2) Mutation efficacy: a longer syscall sequence implies more mutation points (e.g., all arguments in the syscalls); however, not all mutations are valuable, which lowers mutation efficacy. (3) Execution efficiency: longer syscall sequences imply not only more execution time on the syscalls themselves, but also more time spent on coverage tracking, merging, and bug checking. This problem is more severe in DFSes compared to LFSes, due to their higher syscall execution latency. To alleviate the above concerns, it is critical to reduce a test case to a disjoint set of *minimally dependent units* before putting them into the seed pool.

The abstract model for test case reduction in MONARCH is

$$\{test'_1, ..., test'_N\} \leftarrow \text{reduce}(test, cov)$$

where it reduces a test case *test* into multiple minimal units $test'_*$. Syscalls in a minimal unit dependently contribute to the discovery of a piece of new coverage, with the help of tracked code coverage `cov` (see §3.3 for details).

To illustrate this test case reduction model, take a DFS instance with one client and one server under the fault mode as an example. Suppose a test case T uncovers new code coverage. The client slice of T is a sequence of three syscalls [`open`, `read`, `fsetxattr`], which, when executed, triggers per syscall coverage $[cov_{open}, cov_{read}, cov_{xatrr}]$ at the client-side. and whole-test coverage cov_{srv} at the server-side. Each cov_* is a set of IDs representing branches in the DFS codebase. Furthermore, MONARCH notices new coverage when executing the `read` and `fsetxattr` syscall, and denotes the new coverage as cov^*_{read} ($\subseteq cov_{read}$) and cov^*_{xattr} ($\subseteq cov_{xattr}$), respectively, as well as new coverage on the server side for the entire test case denoted as cov^*_{srv}. Had per-syscall coverage not implemented in MONARCH, the per-test case new coverage would be $cov^* = cov^*_{read} \cup cov^*_{xattr} \cup cov^*_{srv}$.

Obviously, removing `open` will likely lead to the disappearance of new coverage, as `read` and `fsetxattr` depend on the `fd` retunred by `open`. Thus, for simplicity, we only showcase two possible reduced test cases, T_1:[`open`, `read`] and T_2:[`open`, `fsetxattr`]. The branch coverage when executing T_1 and T_2 will be tracked by MONARCH and are denoted as $cov^{T_1}_*$ and $cov^{T_2}_*$ respectively. And we further use C_1 and C_2 to denote two interesting cases:

- $C_1 = cov^{T_1}_{read} \supseteq cov^*_{read} \wedge cov^{T_1}_{srv} \supseteq cov^*_{srv}$
- $C_2 = cov^{T_2}_{xattr} \supseteq cov^*_{xattr} \wedge cov^{T_2}_{srv} \supseteq cov^*_{srv}$
- If $C_1 \wedge C_2$, T will be reduced to [T_1, T_2].
- If $C_1 \wedge \neg C_2$, T will be reduced to [T_1, T].
- If $\neg C_1 \wedge C_2$, T will be reduced to [T_2, T].

Syscalls	In-memory state	On-disk state	Fault state	Collected State
`Init state`	i0.dents = [.]	i0.dents = [.]		
`mkdir` `A`	i0.dents = [., A] **i1.dents = [.]**	i0.dents = [.]		

`setxattr` `A` `user.key:val`	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = [user.key:val]
		i0.dents = [.]		

`fault_start` *srv*$_1$ crashes	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = [user.key:val]
		i0.dents = [.]	[**srv$_1$**]	

`removexattr` `A` `user.key`	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = []
		i0.dents = [.]	[*srv*$_1$]	

`fault_stop` *srv*$_1$ recovers	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = []
		i0.dents = [.]		

`getxattr` `user.key`	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = []
		i0.dents = [.]		**i1.xattr = [user.key:val]**

`Final state`	i0.dents = [., A]
	i1.dents = [.]
	i1.xattr = []
		i0.dents = [.]		**i0.dents = [., A]**
				i1.dents = [.]
				i1.xattr = [user.key:val]

Table 3: The symbolic emulation procedure of the SVF in Figure 4. Here, empty fault states indicate no node faults, and empty collected states imply no data or metadata reading. We omit basic attributes and file data in this example. `fault_start` and `fault_stop` are `fault_start_barrier_c` and `fault_stop_barrier_c`.

In other cases, `read` and `fsetxattr` syscalls might somehow interfere on the DFS and hence T cannot be reduced.

3.5 Memory and Semantic Checker

In this subsection, we apply the distributed bug-checking logic introduced in §3.1 to the memory and semantic checker.

Memory checker. `Checker` reports bugs to users once it receives a crash notification from any `CheckerAgents` which monitor process crashes on each node as the bug oracle. For instance, dynamic memory bug detectors like ASan [55] and KASan [33] will crash processes or kernels when memory bugs are found with a detailed report on the memory error.

Semantic checker. We propose a semantic checker—SYMSC—that symbolically emulates syscalls to detect four kinds of semantic bugs. SYMSC compares its generated symbolic states with the collected runtime states for each test case to report semantic bugs. SYMSC maintains the symbolic representation of DFS states, including in-memory states, persistent states, and fault states. The in-memory and persistent states represent files and directories in a DFS instance through inodes with their basic inode attributes (*e.g.*, mode), extended attributes, file data, and directory entries, whereas fault states track node crashes and network partitions.

Table 3 presents the symbolic execution of the SVF shown in Figure 4. Initially, there is only a root directory in the in-memory and on-disk state. Then, SYMSC updates these states by symbolically executing the syscall on the current state according to POSIX and the semantics of pseudo syscalls. For example, `mkdir` and `setxattr` update the in-memory state by adding an inode `i1` and setting an extended attribute "`user.key:val`" on it. Then, the fault state for srv_1 is updated when it is down and recovered by `fault_start` and `fault_stop`. SymSC asserts bugs once there is an inconsistency between the symbolic states and the runtime states collected by `CheckerAgents`. The runtime states include (1) data/metadata returned by syscalls (*e.g.*, `getxattr` returns "`user.key:val`") and (2) data/metadata of every file and directory, within the file system instance, extracted after an execution. The final emulated state, indicated by the last row and second column, reveals that the extended attribute "`user.key:val`" should not exist. However, it is still accessible in the collected runtime state (last column), prompting SymSC to identify this as a bug.

However, the aforementioned state transition still falls short in DFSes due to three main issues. We enhance the state transition to address these issues step by step below.

Issue 1: *DFSes might not comply to POSIX spec strictly.* For instance, POSIX mandates strong consistency, ensuring every read accesses the latest writes, whereas `NFS` employs close-to-open consistency, only guaranteeing that a client can read the data written by another one after the file is closed on that client.

Solution 1: *SymSC is adjusted according to the customization over POSIX specifications.* For example, to address the customized consistency model (*i.e.*, close-to-open consistency) in `NFS`, SymSC maintains multiple versions of file data and metadata. It asserts semantic correctness by checking if the retrieved runtime data or metadata is one of the versions between `open` and `close`. Additionally, when a `GlusterFS` client creates a file `A`, servers create `A′` with the same name on their underlying LFS in specific configurations. Moreover, the metadata of `A` is stored as extended attributes of `A′`. Users can retrieve these server-set attributes on `A′` by executing `getxattr` on `A`, violating POSIX specification as users do not set these attributes and therefore should not be able to retrieve them. Obviously, developers know it and customize the behavior of `getxattr`. To integrate this customization, we filter out server-set attributes from runtime states in SymSC when comparing emulated and runtime states.

Issue 2: *POSIX spec does not consider fault scenarios.* Unlike LFSes, the (in-memory and on-disk) state transition in file system given a syscall is not deterministic in DFSes. As the example shows in Figure 4 in §2, syscalls can operate normally if the fault state of DFS cluster satisfies certain criteria (named fault model).

Solution 2: *Extending POSIX spec with DFS fault models*

Given a fault state and a syscall C on client N_i, if the DFS is available for C at N_i according to the fault model, we apply the POSIX semantics on C, otherwise, C returns an error.

We summarize fault models from DFSes and encode them into SymSC. The exacted fault models can be generalized into data/metadata distribution and availability mechanism. The former decides involved nodes when issuing a syscall, while the latter tells if syscalls can operate normally based on the states of involved nodes. Generally, DFSes adopt consistent-hashing-alike algorithms to distribute data/metadata, and quorum mechanism as availability conditions.

Issue 3: *POSIX spec does not specify concurrency behaviors.* When syscalls are issued from multiple processes or client nodes concurrently, their execution periods are either **disjoint** or **overlapping**. Disjoint syscalls are sequentially orderable based on their invocation and return timestamps, indicating a specific execution sequence. Conversely, overlapping syscalls are not, because their overlapping timestamps prevent inferring the actual execution order of inside DFSes. *Overall, concurrent syscalls are partially ordered.* [2] Taking the example below, assume two concurrent `append` operations on the same file, `"OP1"` and `"OP2"`, are issued by two clients, writing data `"AA"` and `"BB"` (2 bytes) respectively, in a DFS with an atomic write size of 1 byte. If `"OP1"` returns before `"OP2"` is invoked, they are disjoint syscalls. Thus, they are ordered, and the file data is `"AABB"`. On the contrary, if `"OP2"` is invoked after `"OP1"` is invoked but before its return, there is no clearly defined order between them according to the timestamps. Therefore, they can interleave at the granularity of atomic operations and produce six possible file data as follows: `"AABB"`, `"ABAB"`, `"ABBA"`, `"BBAA"`, `"BABA"`, or `"BAAB"`. For example, `CephFS` allows the write atomicity at 4 MB granularity [5]. Notably, some syscalls themselves are atomic operations, such as `chmod`.

Solution 3: *Extending POSIX with atomicity guarantees.*

For overlapping syscalls, any interleavings among their atomic operations are allowed.

To construct the partial order relation among syscalls, `CheckerAgents` record the timestamp from a clock that is synchronized to all distributed nodes, before and after the execution of each syscall, respectively. Thanks to our VM-backed design running on one physical machine, we get a monotonically increasing global clock across all the VMs using the physical timestamp counter. Moreover, we extract the atomic operations from POSIX and DFS documentation and encode them within SymSC.

SymSC explores all possible interleavings among the atomic operations associated with overlapping syscalls. The exploration process ends when either SymSC finds an interleaving that aligns with the observed runtime state, or it fails to find one, which results in a semantic bug. We further prune the interleaving space with dynamic partial order reduction (DPOR) [18]. DPOR reduces redundant states by defining a set of independent events, whose order does not affect the execution result. In our case, syscalls are defined as events. We define concurrent syscalls to be independent

[2]For a given set of events, if any two events can be ordered, they are totally ordered. Otherwise, they are partially ordered.

Component	LoC	Language
Compilation & DFS configuration		
DFS compiler wrapper	205	C++
DFS configuration scripts	755	bash
Monarch Framework		
Fuzzer	5,213	Golang
Executor	600	C++
Coverage collection in userspace (Ucov)	683	C++
Checker & Checker agent	1,594	C++/python

Table 4: Implementation complexity of Monarch.

based on the following conditions: 1) syscalls on distinct files; 2) only data/metadata reads on files / directories; and 3) non-conflicting reads and writes on files / directories.

4 Implementation

We implement Monarch based on Syzkaller [16] to reuse its features like Kcov [32] and syscall mutations. Further, we use QEMU `ivshmem`, an inter-VM shared memory device for communication among VM-backed nodes, such as distributing test cases, collecting coverage, and synchronizations. The complexity of each component is shown in Table 4. The remaining components are illustrated below.

Code coverage collection. Coverage tracking logic is instrumented via the `-fsanitize-address` flag in GCC. Monarch only instruments the source code of DFSes and not other code involved (e.g., `ext4`). Coverage for kernel components is tracked via *Kcov* [32] while our home-grown *Ucov* is used to track coverage in userspace components.

Faults. Fault injections are implemented as pseudo syscalls within Syzkaller, which are just normal user space functions but can be integrated into syscall sequences. Specifically, the node crashes are implemented via `sysrq`, while network partitions are realized through `iptables`.

Synchronization primitives. Multiple bits in the `ivshmem` are reserved to represent the states of injected faults, allowing Monarch to implement pseudo syscalls as listed in Table 2.

Checker. We employ ASan [55] and KAsan [33] for memory bugs. And we implement SymSC based on SymC3 [34]. To achieve reading the physical TimeStamp Counter (TSC) from VM guests, we disable VM-exits when executing `rdtsc` instructions and set the TSC offset as zero, which hardware adds to the physical TSC and further returns to the guests.

5 Evaluation

We evaluate Monarch on six DFSes with the goal of answering the following questions.

Q1. How effective is Monarch in finding bugs? (§5.1)

Q2. What are the practical execution state representations in DFSes? (§5.2)

Q3. How are the fuzzing speed and semantic checker performance? Does reduction improve performance? (§5.3)

DFS	Memory Bugs		Semantic Bugs						
			SVM		SVP	SVF	SVC	
	#C/#D	#FC	#C/#D	#E	#C/#D	#E	#C/#D	#E
`Lustre`	4 / 8	0	0 / 0	0	0 / 0	0	0 / 0	0
`GlusterFS`	3 / 17	12	1 / 1	2	1 / 1	1	3 / 3	2
`OrangeFS`	0 / 3	2	0 / 0	0	0 / 0	0	0 / 0	0
`BeeGFS`	0 / 0	0	0 / 0	0	2 / 2	0	0 / 0	0
`CephFS`	3 / 4	0	0 / 0	0	0 / 0	0	1 / 1	0
`NFS`	1 / 8	0	0 / 0	0	0 / 0	0	0 / 0	0
Total	11 / 40	14	1 / 1	2	3 / 3	1	4 / 4	2

C: **C**onfirmed new bugs **D**: **D**etected new bugs
FC: New bugs triggered by **F**aults and **C**alls **E**: **E**xisting bugs

Table 5: Monarch found **48 new bugs** and **5 existing bugs** in the above six DFSes in every bug category.

Experiment setup. We evaluate Monarch on two 64-core machines with AMD EPYC 9554 Processor and 755GB memory. The fuzzing targets consist of six popular DFSes listed in Table 5. For the evaluations, Monarch deploys VMs of each fuzzing instance on the same physical machine, and initializes the corpora as empty when starting fuzzing.

5.1 Bug-Finding Result

New bugs found. After running on six DFSes intermittently for two months, Monarch discovered bugs in every single DFS and 48 bugs in total as shown in all columns *#D* of Table 5, including 40 memory bugs and eight semantic bugs. Additionally, Monarch did not produce any false positives during our evaluation. DFS developers have confirmed 19 out of 48 bugs. The remaining 29 memory bugs are still under confirmation as developers require deterministic reproducing steps for acknowledgments. While Monarch frequently triggers these bugs, their complex concurrency inside DFSes demands significant manual effort to deduce deterministic reproduction steps. Although not confirmed, these are real bugs because ASAN [55] and KASAN do not produce false positives. Interestingly, although some bugs appear deceptively simple on the surface, they have existed for a long time. For example, a SVP (crash consistency bug) in `GlusterFS` has existed since its initial release. The results show that Monarch is effective in finding bugs in popular POSIX-compliant DFSes even if they have already been tested using rigorous regression testing and comprehensive unit testing.

Bug characteristics. We now summarize the characteristics of new bugs, found by Monarch, below:

- *Faults play a critical role in triggering these bugs.* 14 out of 40 memory bugs require both faults and syscalls as trigger conditions. Moreover, we find three SVP (crash consistency bugs), one in `GlusterFS`, and two in `BeeGFS`, which also involve node crash faults. Similarly, in the existing bugs (illustrated later), the fault, *e.g.*, crashing partial nodes in `GlusterFS`, can lead to one SVF (colum *SVF #E* in Table 5).

- *Vulnerable codes are scattered in both server and client components.* For example, nine memory crashes exposed in DFS servers, while the remaining occur in clients. Moreover, all semantic bugs originate from the vulnerable server code. This characteristic underscores the importance of testing each DFS component and the need for distributed bug-checking logic that can detect bugs across all nodes comprehensively.
- *Exposures of new bugs might depend on the DFS configuration.* The configuration includes file system modes (*e.g.*, distributed, replicated modes in GlusterFS [12]), the number of servers and clients. For example, the memory bug we show in Figure 1 requires a GlusterFS instance configured with distributed mode and two server nodes. Additionally, it requires a fault injected into a specific instead of a random node. Moreover, the semantic bug illustrated in Figure 5 has to be exposed by the concurrent execution from two clients rather than one client.
- *Bugs in DFSes can lead to various consequences.* The new bugs Monarch found can potentially lead to data loss, denial-of-service, potential file system information leakage, and privilege escalation.

Evaluation of SymSC on existing bugs. After reviewing 5,900 bug reports and 6,100 git commits filtered by keywords across six DFSes, we *confidently* identified 20 semantic bugs; however, this does not suggest that only 20 bugs exist among the 5,900 reports. Unlike memory bugs, identifiable through indications of a memory crash, others lack detailed descriptions, preventing classification. Further, only five out of them can be reproduced manually as most of them miss detailed reproducible steps. We show the reproduced five bugs in all columns *#E* in Table 5 and detail them in Table 6. Monarch can reproduce all of them by applying SymSC, which further indicates the effectiveness of Monarch.

Comparison with the state-of-the-art LFS fuzzers. Hydra [34] and Syzkaller [16] are two state-of-the-art LFS fuzzers. Hydra uses a library OS that gets linked with a fuzzing target. Thus, it cannot test DFSes because most DFSes have multiples of user space processes even for one client. Meanwhile, Syzkaller only supports fuzzing NFS. Unfortunately, it could not detect any bugs reported by Monarch. Further, if we enable our multi-node fuzzing architecture but not Ucov on Syzkaller (named Monarch-Basic), it can fuzz DFSes that have in-kernel modules (*i.e.*, NFS, CephFS, and Lustre). It could only find 20 memory bugs. We categorize the remaining bugs that Monarch-Basic cannot find into the following groups: *(1) Fault as an input space*: 17 memory and semantic bugs involve faults. However, Monarch-Basic is incapable of injecting faults into DFSes, and thus misses them. *(2) Cross-node and cross-context coverage*: 21 memory bugs occur in the user-space components of DFSes, which Monarch-Basic cannot detect as it can only test kernel components. *(3) Semantic checkers*: Monarch reports eight semantic bugs, which Monarch-Basic cannot find due to the lack of our semantic checkers SymSC.[3]

Type	Description
SVC	In a GlusterFS cluster, after client 1 opens an existing file "A" successfully and get a returned fd1. The open is cached locally and not sent to the server. Afterward, client 2 deletes "A" with unlink. Since the open is not sent to servers, servers are aware of the file references and thus delete it from the DFS. Finally, a fstat with fd1 on "A" from client 1 results in an ENOENT or ESTALE error. However, according to POSIX specifications, operations on an opened file descriptor should function normally until the descriptor is closed [27].
SVC	Under specific configurations, GlusterFS does not invalidate the client cache correctly, leading to the data written from one client not being seen by another [46].
SVF	File system state modification (*i.e.*, extended attributes) during the offline of a server is not synchronized to it after it is recovered, leading to that clients can read stale data from it [51].
SVM	When creating a file, atime and mtime are updated with the timestamp obtained from the Linux, and ctime is updated with the timestamp from the userspace client modules. [10] Thus, they are inconsistent, which violates the POSIX specification [28].
SVM	GlusterFS does not support O_PATH flag in open [26].

Table 6: Collected existing semantic bugs.

5.2 DFS Execution State Representation

To find out the most practical approach to representing execution states, we compare the finally achieved coverage when the execution states in Monarch are represented by 1) both server and client coverage (client+server); 2) server coverage only (server); 3) client coverage only (client); Regarding each representation, we run them for three rounds with a duration of 48 hours for each round. If the coverage is not converged after 48 hours, we extend the testing duration in increments of 12 hours until it converges. For all except NFS, we set up DFSes with three servers and one client, as Lustre, BeeGFS, and CephFS each need a management server, a metadata server, and a data server. For consistency, we also configure OrangeFS and GlusterFS with three servers, despite their ability to operate with just one server. After averaging the coverage from three rounds, the results of fault and non-fault modes are shown in Figure 8.

Non-fault mode. As depicted in the *first* row of Figure 8, in the non-fault mode, representing the execution states with client coverage yields coverage rates comparable to the representation with both server and client coverage, which matches our intuition *"Finding 1"* in §3.3. However, when representing execution states with server coverage only, the achieved coverage is significantly lower than others. It is because 1) server coverage is less fine-grained than client coverage. This would result in fewer small seeds and further decrease mutation efficiency and fuzzing speed, ultimately

[3]Bugs are counted multiple times across these three categories, as uncovering a bug might require various conditions.

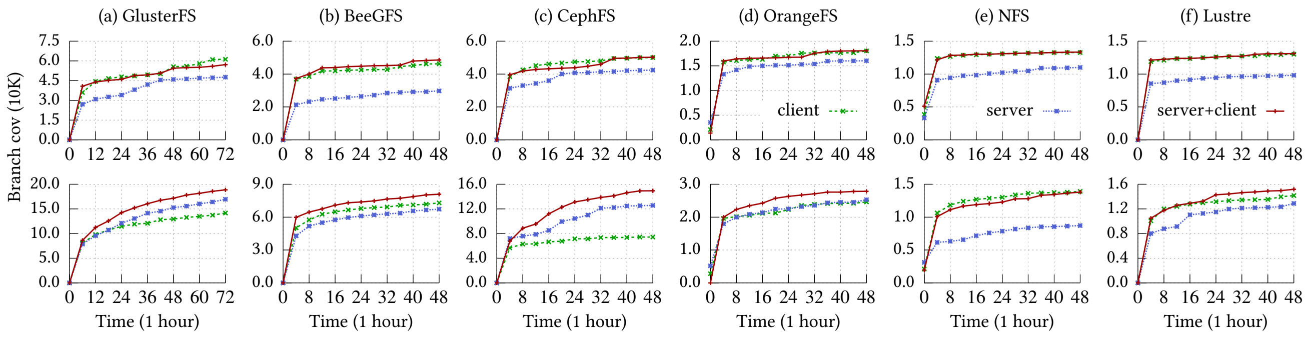

Figure 8: Code coverage growth trend when representing DFS execution states with coverage from **client** + **server**, **client only**, and **server only** in *non-fault* mode (the first row) and *fault* mode (the second row).

leading to lower coverage. 2) Servers cannot fully represent DFS states compared with clients (See §3.3 *"Finding 1"*).

Fault mode. In contrast to the non-fault mode, in fault mode (the *second* row in Figure 8), representing execution states with both server and client coverage yields up to 101% higher coverage than representing the state with either client coverage alone or server coverage alone, such as (c) `CephFS` and (e) `NFS` in the second row of Figure 8. This result is consistent with our intuition that fault injection would activate fault tolerance components on servers, that can only be captured through server coverage and not client coverage.

Furthermore, by comparing the achieved coverage in two rows in Figure 8, faults have the potential to increase code coverage up to three times, e.g., 61K (non-fault) VS 188K (fault) in GlusterFS. The codebase sizes and complexities of DFSes determines how the coverage increases. For example, `CephFS` (931K), `GlusterFS` (556K), and `BeeGFS` (554K) have larger code sizes and more complex fault tolerance mechanism compared to `NFS` (94K) and `OrangeFS` (360K). Therefore, the former three DFSes achieve higher code coverage increase when injecting faults.

Comparison with LFS fuzzers. Syzkaller and Hydra do not support DFSes, but Syzkaller offers a simple interface for `NFS`. Therefore, we only compare Syzkaller and Monarch on `NFS`. Unfortunately, Syzkaller covers 99% less code than Monarch because it does not set up `NFS` servers. Further, we enable the server setup in Syzkaller, but coverage does not increase because Syzkaller's mount-during-test model struggles to generate valid arguments for `syz_mount_image$nfs`, leading to mount failures. Finally, we replace its mount-during-test model with Monarch's mount-before-test model. Syzkaller then covers 13,200 branches, close to Monarch's 13,278. However, the modified Syzkaller still cannot test NFS with injected faults, a feature unique to Monarch.

5.3 Performance

Fuzzing speed. The fuzzing speed in non-fault mode varies from 3 to 15 executions per second across different DFSes, which conforms to the natural performance difference

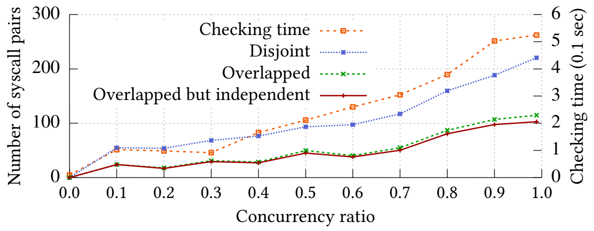

Figure 9: We group test cases by their concurrency ratio (x-axis) and calculate the average number of overlapping, disjoint, and independent but overlapping syscall pairs(left y-axis), along with the average semantic checking time (right y-axis), inside each group.

among DFSes. The fuzzing speed is not as fast as LFS fuzzing (*e.g.*, around 30 exec/sec in Syzkaller), but it makes sense considering DFSes need to synchronize states across nodes through network communication, inherently causing higher latency than LFSes. Further, if we enable fault injections, the fuzzing speed decreases to around 2 execs/sec, as it is time-consuming to launch and recover from faults. We will explore techniques for improving the fuzzing speed of distributed systems in the future, such as optimizing the network stacks under testing scenarios and fast fault injection and recovery.

Performance improvement from test case reduction. To show that test case reduction improves execution efficiency and mutation efficacy as mentioned in §3.4, we run Monarch on six DFSes by enabling and disabling test case reduction and set the maximum number of syscalls per test case as 100. Other configurations are the same as §5.2. When enabling test case reduction, Monarch exhibits a noteworthy 75% reduction in execution time and 93% less syscalls per test case at least, showing the improvements in execution efficiency. Further, coverage, as a symptom of mutation efficacy, is improved in the range of 1% to 31% among six DFSes. This can be attributed to the reduction of syscalls per test case, which subsequently reduces the non-valuable mutation space and thus advances the mutation efficacy.

Semantic checker performance. The length of each syscall sequence and their difference within test cases dominate the number of overlapping syscalls, and further, both affect the semantic checking time. Therefore, it is less informative to simply average these numbers from test cases. Instead, we group test cases according to their concurrency ratio calculated by the following formula, further average the evaluation numbers in each group.

$$ratio = (\sum_{i=1}^{n} t_i / MAX) * (1 - \min_{1 \le i < j \le n} |t_j - t_i| / \sum_{i=1}^{n} t_i)$$

where, t_i is the number of syscalls in the i-th syscall sequence of a test case, and MAX is the predefined upper limit for the number of syscalls in a test case. This formula is based on the intuition that test cases with more syscalls and smaller variance in the length of their syscall sequences should have a higher concurrency ratio.

Figure 9 shows the number of disjoint, overlapping syscall pairs, overlapping but independent syscall pairs, and the semantic checking time in each concurrency ratio. From the result, we can see that as the concurrency ratio increases, (1) the number of disjoint and overlapping syscall pairs increases, but the space of interleaving exploration (*i.e.*, the difference between overlapping and overlapping but independent syscall pairs) keeps almost constant; (2) the time for semantic checking increases due to the length of syscall sequences instead of the number of interleaving exploration space, but it always finishes in around half a second.

6 Discussion

Being the first fuzzer for DFSes, Monarch is far from complete. In this seciton, we discuss the limitation of Monarch and promising future directions.

Scheduling as an input source. Although Monarch can execute syscalls concurrently to find concurrency bugs, it does not explicitly schedule threads in clients and servers, leading to incomplete exploration of the interleaving space and thus bug misses. Additionally, even finding concurrency bugs, it cannot provide sufficient details for deterministic reproduction, which impedes the bug-fix process. We will extend Monarch in this direction.

Porting to other distributed systems. Though specifically designed for POSIX-compliant DFSes, Monarch can be adapted to fuzz other distributed systems, such as databases (e.g., Cassandra [19]), and object storage (e.g., Ceph RADOS [62]), as well as non-POSIX file systems (e.g., HDFS [57]), by plugging-in their API mutator and checkers.

7 Related Work

Bug-finding techniques in distributed systems. Model checking, such as Modist [66] and SAMC [36], enumerate event orderings (e.g., user operations and faults) to expose bugs. Even after adopting partial order reductions to reduce the enumeration space, they still face the state explosion problem in complex cases. Another technique is formal verification [24, 63], which expects users to provide domain-specific knowledge, thereby impeding its popularity and efficiency as a testing method in practice. Dynamic testing, such as Jepsen [60], is also widely adopted, especially in the distributed database area. Compared with Jepsen, Monarch is unique in three ways: *1) Automatic testing from both explicit test cases and implicit faults*: Jepsen is a pure fault injection tool, in which users must provide manually constructed test cases and fault injection schemes to explore the *fault space.* In contrast, Monarch automatically produces both syscall sequences and faults based on *coverage-guided mutation.* It does not require any manual effort from users during testing. *2) Cross-context testing*: Jepsen exclusively targets userspace applications (*i.e.*, MongoDB and Redis), while Monarch, being more comprehensive, handles both in-kernel and userspace distributed systems. *3) Semantic checker tailored for DFSes*: Monarch is equipped with our DFS checker—SYMSC—that detects unique DFS semantic bugs (see §2.2 and §3.5). However, Jepsen only provides the database transactional consistency checker Elle [35], which does not apply to DFSes.

Fault injection in distributed systems. Most existing works inject faults at points specified by users [3, 60] or heuristics [11, 23, 38, 39]. Molly [4] and Mallory [42] advanced these works by utilizing lineage-driven and timeline-driven, respectively, to inject faults adaptively. However, they concentrate on fault injection for specific user inputs, rather than automatically exploring the execution state space of distributed systems from both explicit user inputs and faults.

8 Conclusion

This paper introduces Monarch, the first multi-node fuzzing framework for finding memory and semantic bugs in POSIX-compliant distributed file systems, through the following novel designs: 1) Testing all components of DFSes holistically; 2) Testing from both explicit inputs "syscalls" and implicit inputs "faults" with a two-step mutator; 3) Adopting practical execution state representations for fault and non-fault testing modes by measuring different choices, and proposing a unified coverage collection scheme for both user and kernel contexts; 4) Designing a semantic checker SymSC for capturing all types of semantic bugs in DFSes. So far, Monarch has identified 48 bugs in six popular DFSes.

Acknowledgments

We thank the anonymous shepherd and reviewers for their helpful feedback. We also thank Manuel Egele and Gwangmu Lee for joining the project discussion at the early stage. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868) and SNSF PCEGP2_186974.

References

[1] Ace computers, beegfs streamline cluster workflow for elite defense contractor. https://www.beegfs.io/docs/flyers/Ace%20Computers-Case%20Study-Defense%20Contractor-BeeGFS.pdf.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[3] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.

[4] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, Melbourne, Victoria, Australia, May 2015.

[5] Ceph authors and contributors. Ceph posix compatibility. https://docs.ceph.com/en/reef/cephfs/posix.

[6] Ceph authors and contributors. Ceph unit tests. https://docs.ceph.com/en/quincy/dev/developer_guide/tests-unit-tests/.

[7] OrangeFS authors. The orangefs project. http://www.orangefs.org/.

[8] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[9] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[10] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[11] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: consistency-guided fault injection for cloud systems. In *Proceedings of the 35rd International Conference on Computer Aided Verification (CAV)*, Virtual Event, Australia, September 2020.

[12] Gluster community. Glusterfs. https://docs.gluster.org/en/main/Quick-Start-Guide/Architecture/.

[13] Gluster community. Glusterfs replica 3 volume and quorum. https://docs.gluster.org/en/main/Administrator-Guide/Split-brain-and-ways-to-deal-with-it/#client-quorum-in-replica-2-volumes.

[14] Gluster community. Regression testing in glusterfs. https://docs.gluster.org/en/main/Developer-guide/Development-Workflow/#auto-triggered-tests.

[15] Jonathan Corbet. The curious case of o_directory|o_creat. https://lwn.net/Articles/926782/.

[16] David Drysdale. Coverage-guided kernel fuzzing with syzkaller, 2016.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[18] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005.

[19] Apache Software Foundation. Apache cassandra. https://cassandra.apache.org/_/index.html.

[20] ThinkParQ GmbH. Beegfs. https://www.beegfs.io/.

[21] Google. Kernelmemorysanitizer, a detector of uses of uninitialized memory in the linux kernel. https://github.com/google/kmsan.

[22] Google. Syzlang: syscall description language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[23] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, USA, March 2011.

[24] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[25] Christoph Hellwig. Xfs: the big storage file system for linux. *The magazine of USENIX & SAGE*, 34(5):10–18, 2009.

[26] Xavi Hernandez, Amar Tumballi, , and Mohit. Glusterfs doesn't support o_path flag in open. https://github.com/gluster/glusterfs/issues/2717.

[27] Xavi Hernandez, Amar Tumballi, and Mohit. Open-behind should be disabled by default. https://github.com/gluster/glusterfs/issues/3785.

[28] Kotresh HR. Ctime: Fix ctime inconsisteny with utimensat. https://github.com/gluster/glusterfs/commit/1dec1d6.

[29] IEEE. Posix specification. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[30] Internet Engineering Task Force (IETF). Network file system (nfs) version 4 protocol. https://www.rfc-editor.org/rfc/rfc7530.html.

[31] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2104–2121. IEEE Computer Society, 2023.

[32] The kernel development community. Kcov: code coverage for fuzzing. https://docs.kernel.org/dev-tools/kcov.html.

[33] The kernel development community. The kernel address sanitizer (kasan). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[34] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[35] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[36] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[37] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[38] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, March 2018.

[39] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[40] Sage McTaggart. Cve-2018-10927. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10927.

[41] Sage McTaggart. Cve-2022-3650. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3650.

[42] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, November 2023.

[43] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.

[44] OpenSFS and EOFS. Lustre. https://www.lustre.org/.

[45] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[46] Philipspencer and Mohit. Fuse mount cache invalidation does not work with stat-prefetch disabled. https://github.com/gluster/glusterfs/issues/4281.

[47] Lyu, Tao and Karampuri, Kumar Pranith. A semantic violation on in-memory states (SVM) in GlusterFS. https://github.com/gluster/glusterfs/issues/3624.

[48] Lyu, Tao and Karampuri, Kumar Pranith. A semantic violation on persistent states (SVP) in GlusterFS. https://github.com/gluster/glusterfs/issues/3983.

[49] Lyu, Tao and Li, Xiubo. A semantic violation under concurrent executions (SVC) in CephFS. https://tracker.ceph.com/issues/63906.

[50] Lyu, Tao and Mohit. A use-after-free bug in GlusterFS. https://github.com/gluster/glusterfs/issues/3732.

[51] Rofman, Sason Barak and Mohit and Karampuri, Kumar Pranith. A semantic violation under fault states (SVF) in GlusterFS. https://github.com/gluster/glusterfs/issues/1324.

[52] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[53] Pedro Sampaio. Cve-2018-1088. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1088.

[54] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, RENNES, France, April 2022.

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[56] Amazon Web Services. Using amazon fsx for lustre for genomics workflows on aws, 2020. https://aws.amazon.com/blogs/storage/using-amazon-fsx-for-lustre-for-genomics-workflows-on-aws.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[58] Andriy Skulysh. Rpc resend may corrupt the data. https://jira.whamcloud.com/browse/LU-11444?jql=text%20~%20%22data%20corruption%22.

[59] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[60] Jepsen team. Jepsen. https://jepsen.io/.

[61] PyTorch team. Training a 1 trillion parameter model with pytorch fully sharded data parallel on aws. https://medium.com/pytorch/training-a-1-trillion-parameter-model-with-pytorch-fully-sharded-data-parallel-on-aws-3ac13aa96cff.

[62] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[63] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 2015.

[64] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[65] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[66] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2009.

[67] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[68] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. Syzscope: Revealing high-risk security impacts of fuzzer-exposed bugs in linux kernel. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.