Themis: Finding Imbalance Failures in Distributed File Systems via a Load Variance Model

Yuanliang Chen
KLISS, BNRist, School of Software, Tsinghua University
China

Fuchen Ma*
KLISS, BNRist, School of Software, Tsinghua University
China

Yuanhang Zhou
KLISS, BNRist, School of Software, Tsinghua University
China

Zhen Yan
KLISS, BNRist, School of Software, Tsinghua University
China

Qing Liao
Harbin Institute of Technology, Harbin, Heilongjiang
China

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

Abstract

A distributed file system (DFS) is a file system that spans across multiple file servers or multiple locations. The load balancing mechanism in a DFS is crucial, as it optimizes resource utilization across all nodes and improves response times. However, incorrect load scheduling or implementation errors in load balancing algorithms can lead to system imbalance, hang-ups, and even crashes. Such imbalance failures may be critical and pose a significant threat to the availability and security of distributed file systems.

This paper presents a detailed study of real-world imbalance failures in four widely used DFSes, exploring their symptoms and triggering conditions. We found that test cases that incorporate both client requests and system configuration inputs are crucial for exposing these imbalances. However, generating such high-quality test cases is challenging due to the extensive combinations of these two input spaces. Guided by our study, we designed a testing framework named Themis. To efficiently prune the search space, Themis first models both the request and configuration inputs and transforms them into operation sequences. It then employs load variance-guided fuzzing to thoroughly explore the operation sequence and constantly generate test cases that make nodes loaded as differently as possible. Finally, Themis introduces a load detector to monitor the resource usage of each distributed node and precisely identify any imbalances. Themis has detected 10 new imbalance failures in four real-world DFSes, which have been addressed by the respective maintainers.

CCS Concepts: • **Security and privacy** → *Database and storage security.*

Keywords: Testing, Distributed File System, Load Balance

ACM Reference Format:
Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Zhen Yan, Qing Liao, and Yu Jiang. 2025. Themis: Finding Imbalance Failures in Distributed File Systems via a Load Variance Model. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696082

1 Introduction

Distributed file systems (DFSes) have become ubiquitous in modern computing environments, driven by the need for scalable storage solutions [27, 42]. As organizations increasingly rely on DFSes to manage vast data volumes across interconnected nodes, the importance of efficient load balancing mechanisms has been emphasized [28, 29]. Evenly distributing data and processing loads is crucial for optimizing resource utilization and ensuring high availability, scalability, and reliability. Load balancing mechanisms dynamically allocate and redistribute tasks to prevent any single node from becoming a bottleneck, improving system performance, scalability, and reliability.

Load balancing mechanisms are inherently complex, with intricate architectures that manage data across multiple dynamically changing nodes. Consequently, it is difficult to avoid implementation errors in these mechanisms. Given the critical role these mechanisms play in DFSes [13], errors causing imbalanced load distribution or 'hot spots' can lead to severe consequences [9], including degraded performance and potential service unavailability, thereby impacting the reliability, availability, and security of DFSes. For instance,

*Fuchen Ma and Yu Jiang are the corresponding authors.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696082

in 2018, GitHub faced a network partition outage lasting over 24 hours, caused by improper load distribution and subsequent system overload, which affected millions of developers worldwide. Similarly, due to load balancing errors, Virgin Blue [36] experienced an 11-day outage, while Microsoft Azure Storage Service was interrupted [25], causing severe financial losses. Such failures, caused by errors in load balancing mechanisms that result in disproportionate load distribution, leading to hot spots, hang-ups, or even crashes, are what we refer to as **Imbalance Failures**.

We first conduct a comprehensive study on imbalance failures based on 53 real-world imbalance failures from four widely used DFSes, including HDFS [55], CephFS [59], GlusterFS [20], and LeoFS [41]. For each failure case, we manually analyzed their reports and the corresponding fixes to thoroughly understand their symptom severity and triggering conditions. Our study revealed two key insights: ① Most imbalance failures (82%) can lead to significant issues that affect all or a majority of distributed nodes; ② While only a few imbalance failures are triggered by straightforward scenarios, i.e., either through client requests (e.g., file write/read) or system configuration (e.g., volume expansion, node scaling) alone, most (83%) require a combination of both request and configuration inputs to trigger.

Researchers have proposed various technologies for detecting failures in DFSes. Workload generation tools like SmallFile [2] and Filebench [18] excel at generating distributed workloads for DFSes testing but primarily focus on file-related operations, neglecting the impact of system configuration changes on load balancing. Traditional file system testing tools such as Janus [62] and Hydra [38] explore file operation and file image inputs alternately, successfully identifying numerous implementation errors in file systems. Distributed system testing tools such as CrashFuzz [19] and Mallory [45] primarily focus on common issues such as error handling or logic bugs, which typically involve exploring only the fault input space. However, most imbalance failures require coordinated interactions between configuration and request inputs to be triggered, a scenario existing tools struggle with due to their limited exploration of the potential execution dependencies between these input spaces. Additionally, all existing testing tools lack a precise detector for imbalance failure detection in DFSes.

To effectively detect imbalance failures in distributed systems, there are two main challenges: (1) **The first challenge** is to develop a precise detector that can identify whether loads in DFSes are imbalanced. Load distribution varies across different DFSes and changes dynamically. Even though failures occur, it is difficult to detect them without a precise imbalance detector. (2) **The second challenge** lies in generating high-quality test cases for efficiently triggering the imbalance detector. Some imbalance failures tend to be hidden deep within the system, requiring specific execution dependencies, i.e., distributed nodes executing a series of varied operations involving both client requests and system configurations, to activate them. The vast number of possible combinations between the two types of inputs makes it difficult to effectively explore the input spaces.

To tackle these challenges, we introduce Themis, a testing framework designed to automatically detect imbalance failures in DFSes. Themis first designs a general test case specification that models both client requests and system configuration inputs, converting them into an operation sequence. In this way, Themis simplifies the complexity of handling two separate input spaces by reducing them to a single operation sequence, which is well-suited for exploration using fuzzing techniques [44, 68]. To effectively trigger deep imbalance failures, Themis employs load variance-guided fuzzing, inspired by our study, which found that the ultimate imbalanced state results from many cumulative small storage differences across distributed nodes. Themis continuously generates numerous test cases designed to vary the distributed nodes' load as much as possible. Finally, to precisely identify imbalance failures, we propose an imbalance detector that monitors the load states of distributed nodes.

We implemented Themis and evaluated it on four widely used DFSes: HDFS, CephFS, GlusterFS, and LeoFS. Compared with other state-of-the-art testing approaches, Themis excelled in exposing more imbalance failures and achieved 10% to 21% higher code coverage. Additionally, Themis identified 10 new imbalance failures in total, with 4 in GlusterFS, 3 in LeoFS, 1 in CephFS, and 2 in HDFS.

In summary, we make three key contributions:

- We conducted a detailed analysis of 53 real-world imbalance failures in widely used DFSes and determined their symptom severity and triggering conditions.
- We designed a testing framework that synthesizes high-quality test cases incorporating both request and configuration inputs, effectively detecting imbalance failures through a load variance model.
- We implemented and evaluated Themis on four widely used DFSes. We will open-source Themis[1] for practical usage. Currently, it has detected 10 imbalance failures.

2 Background

2.1 Load Balancing Mechanism in DFS

In a DFS, nodes or volumes can be removed, replaced, or added, and files can be dynamically created, deleted, or appended. All of these operations affect the load distribution within the DFS [13, 42, 56]. The load balancing mechanism ensures that data and processing tasks are evenly distributed across nodes, preventing overburdening and optimizing efficiency. Figure 1 illustrates a typical load balancing mechanism, which includes three main kinds of balancers: network, computation, and storage balancers.

[1]Themis: https://anonymous.4open.science/r/Themis-97C4/

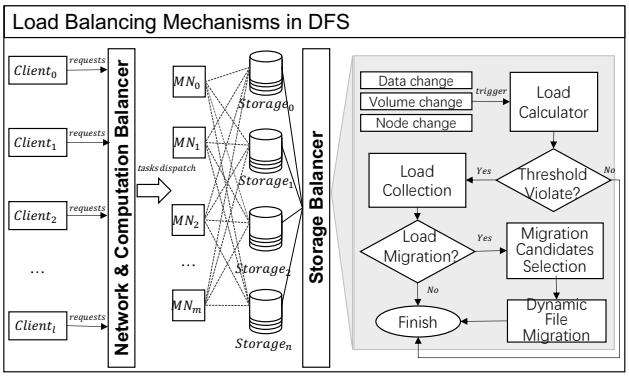

Figure 1. The typical load balancing mechanism in the DFS. MN_i presents the metadata management nodes and $storage_j$ means the storage nodes in the system.

The influx of intensive client requests, encompassing actions like file creation and data searches, etc., can occur rapidly in a DFS. To enhance resource utilization and alleviate performance bottlenecks, various load balancing algorithms, including hash partitioning [61], least connections [1], and weighted distribution [54], are employed to intelligently distribute incoming requests, computation tasks and data storage for nodes in the DFS. For example, consider the storage load balancing process, as shown in Figure 1. In a healthy DFS, storage nodes are primarily responsible for storing file data. The storage load balancing mechanisms [26, 60] are activated in response to various events, such as data changes due to file operations, volume adjustments from expansion or reduction commands, or changes in node dynamics, including entries and exits. In practical load balancing implementation, the load distribution among nodes is typically relatively balanced, not absolutely equal. To this end, a Load Calculator is employed to determine if the DFS is imbalanced by checking whether the current load distribution exceeds a predefined threshold (e.g., the default threshold in the HDFS Balancer [24] is set at 10%). If the threshold is exceeded, the Load Collector recalculates an optimal storage distribution and initiates data migration to achieve a balanced state. Note that since real world DFSes already store a large amount of data, the frequency of triggering a 10% load imbalance is usually quite low, and the overhead caused by data migration is generally minimal. Thus, the load balancing mechanism in DFSes is influenced by two primary types of inputs: **Client Requests** and **System Configurations**.

2.2 Definition of Imbalance Failures

System Model: Throughout this paper, we use the following system model. First, we formally define a DFS as $\phi = \{S, M, L\}$. Specifically, $S = \{s_1, s_2, ..., s_n\}$ represents the set of storage nodes. $M = \{m_1, m_2, ..., m_l\}$ denotes the set of management nodes. $L = \sum(Ls, Ln, Lc)$ represents the total load within the DFS. Here, $Ls = \sum_{i=1}^{n} fd_i$ indicates the total storage load, where fd_i is the file data stored on node s_i. $Ln = \sum_{i=1}^{l} rps_i$ represents the total network load, where rps_i represents the number of requests received per unit time by node m_i. $Lc = \sum_{i=1}^{l} cpu_i$ represents the total computation load, where cpu_i is the CPU resource usage of node m_i.

We assume that all nodes work normally and that the hardware performance between them is very similar, almost indistinguishable (i.e., the difference between nodes' disk read and write capacity, CPU performance, network bandwidth, and latency is small and can be negligible). Formally, we define the Load Balance State (*LBS*) in a DFS as:

$$LBS := \frac{MAX_{a=1...l}\{rps_a\}}{\frac{1}{l}\sum_{a=1}^{l}\{rps_a\}} \le t \;\&\; \frac{MAX_{a=1...l}\{cpu_b\}}{\frac{1}{l}\sum_{b=1}^{l}\{cpu_b\}} \le t \;\&\; \frac{MAX_{c=1...n}\{fd_c\}}{\frac{1}{n}\sum_{c=1}^{n}\{fd_c\}} \le t$$

All types of load $L = \sum(Ls, Ln, Lc)$ in a DFS need to be distributed evenly. This means that any type of load variance between any two nodes should not exceed a threshold value of t. Otherwise, the DFS is considered to be in a Load Imbalanced State. In HDFS Balancer [24], the default threshold value t is 10%. In GlusterFS Balancer [3], the default threshold value t is 20%.

We define **imbalance failures** as errors in the load balancing mechanism of DFSes that can lead ϕ to enter the imbalanced state for an extended period and cannot automatically recover to the *LBS* state. To effectively detect imbalance failures, the main idea is to construct a large number of test cases that bring the DFS into the imbalanced state as much as possible, triggering and exercising the load balancing mechanism more frequently. For a DFS under test, after conducting an imbalance test that triggers the load balancing mechanism and executes corresponding rebalancing code, the system should return to its normal *LBS* state and provide functional services as usual. Otherwise, the DFS remains in the imbalanced state, indicating an imbalance failure.

3 Motivation Study

To better understand the characteristics of imbalance failures in practical scenarios and guide our detection strategy, we selected four widely used DFSes as our study targets: HDFS, CephFS, GlusterFS, and LeoFS. We identified potential imbalance failures by searching the issue trackers of these systems for reports containing the keyword 'balance'. After manually reviewing these reports to exclude issues not related to the load balancing mechanism in DFSes, we identified 53 imbalance failure cases, as presented in Table 1.

Table 1. Number of imbalance failures we analyzed.

HDFS	CephFS	GlusterFS	LeoFS	Total
18	16	12	7	53

3.1 Symptoms of Imbalance Failures

We first examined the consequences and symptoms of each imbalance failure to aid in designing a precise anomaly detector for automatic identification.

Finding 1: imbalance severity. *Most (82%) imbalance failures lead to serious consequences that affect all or a majority of distributed nodes instead of only a few of them.*

We found that 20 out of 53 (38%) imbalance failures lead to performance degradation, significantly slowing down the entire system (e.g., GlusterFS bug #3356 [34]). Additionally, 9 out of 53 (17%) bugs cause partial system outages, making some but not all services unavailable (e.g., HDFS bug #13279 [33]). 7 out of 53 (13%) bugs result in data loss (e.g., LeoFS bug #1115 [48]), while a further 7 out of 53 (13%) bugs can lead to the complete failure of the entire cluster, severely impacting the system's availability and security (e.g., CephFS bug #64333 [10] caused the entire Ceph cluster to crash). The remaining 10 out of 53 (18%) imbalance failures cause either partial request suspensions or single-node performance bottlenecks, affecting only a limited number of nodes and users (e.g., CephFS bug #65806 [63]). Thus, imbalance failures in DFSes are severe and deserve greater attention.

Finding 2: imbalance root cause. *Most (72%) imbalance failures are caused by implementation errors in the data migration process of load balancing mechanisms.*

We then conducted a root cause analysis for each imbalance failure by examining the patches used to fix them. We found that 15% of imbalance failures are triggered by implementation errors in the load calculation processing. For example, HDFS bug #13279 [33] mistakenly includes the load of offline nodes in its calculations, resulting in an imbalanced load distribution plan. Additionally, 13% of imbalance failures result from code mistakes in the load state collection process of DFSes: for instance, in CephFS bug #64611 [57], different daemons return inconsistent status codes, leading to errors during load state collection and consequently causing system load imbalance. The majority of imbalance failures (72%) are caused by implementation errors in the data migration logic: for instance, GlusterFS bug #3513 [7], where improper error handling during the data migration caused load imbalance, eventually leading to data loss.

Finding 3: internal symptoms. *All 53 imbalance failures lead to significant disparities in the usage of computing resources, such as network traffic, CPU, and disk space.*

Although the symptoms of these imbalance failures are subtle, through further analysis of each failure's internal states during the imbalance occurrence, we find that before these imbalance failures lead to serious consequences, there are significant variances in resource usage among distributed nodes. The load disparity between nodes is at least 30%, and in some cases, it exceeds 100%. We discovered that 64% of imbalance failures result in significant disparities in disk usage across storage nodes, 21% cause considerable variations in CPU usage, and 15% lead to substantial differences in network traffic. This finding suggests that monitoring the internal states of the DFS under test, e.g., CPU, storage, network, etc., could be an effective way to identify imbalance failures before they cause serious consequences, e.g., data loss, crashes, etc.

3.2 Triggering of Imbalance Failures

We then analyzed the production steps of each imbalance failure to understand how these failures are triggered, which assisted us in designing an automatic triggering strategy.

Finding 4: triggering workload. *Most (83%) imbalance failures require both client requests and configuration changes.*

We found that a small percentage (13%) of imbalance failures are triggered solely by one type of input, such as client requests for file creation, deletion, and rewriting. A few imbalance failures (4%) can be triggered solely by system configuration changes, like volume expansion and node scaling. However, the majority (83%) of imbalance failures require specific workloads from both request and configuration inputs to be triggered.

Finding 5: triggering steps. *All 53 imbalance failures can be triggered with no more than 10 distributed nodes repeatedly executing short sequences of up to 8 operations, with gradual variation in the operation sequences as they are repeated.*

More than half (66%) of imbalance failures can be triggered in no more than five steps. For example, bug #63014 [67] in CephFS, which is caused by increased latency across multiple mclock queues, resulting in imbalanced traffic, can be exposed through a critical three-step process: restarting the storage node to clear its storage device, writing large amounts of file data to exercise the data indexing, and then monitoring the load distribution of nodes to reveal the bug. Another 34% of imbalance failures are hidden in deeper logic and require sequences of 6 to 8 operations to trigger. For instance, bug #1245142 [47] in GlusterFS, which incorrectly returns a balance status, requires sequences of at least 8 operations: 'create, volume_add, mount, mkdir, touch, remove, node_kill, status' to be triggered successfully.

Finding 6: imbalance accumulation. *The load imbalanced status in DFS is not achieved all one stroke; rather, it accumulates gradually through minor imbalances.*

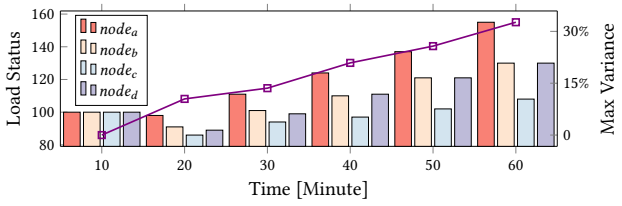

Figure 2. Storage status of each distributed storage node during the reproduction of Bug GlusterFS-3356.

Additionally, we recorded and analyzed load changes during the imbalance failure triggering process. We found that the ultimate load imbalanced state in a DFS is usually caused by continuously accumulating many intermediate states of slight load variances. For example, in bug #3356 [34] in GlusterFS, Figure 2 shows how disk resource usage across nodes

changes during bug reproduction. The line chart indicates the maximum storage variance among the nodes. Throughout the bug reproduction, the disk load variance between $node_a$ and $node_c$ gradually increases until $node_c$ reaches full capacity and becomes a 'hotspot', eventually triggering the bug. This observation suggests that we can use runtime load variance as feedback to guide optimized test case generation.

3.3 A Motivation Example

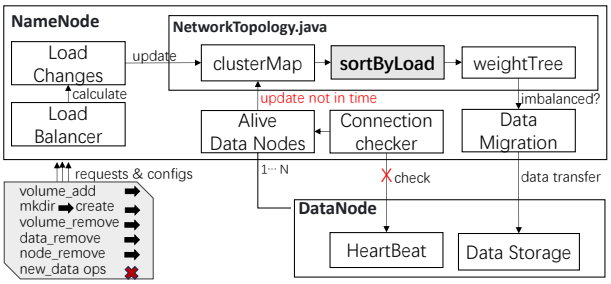

Figure 3. The HDFS-13279 imbalance failure causes storage hotspots and blocks new data from being stored in the DataNodes, resulting in service unavailability for HDFS.

```
1  private void sortByLoad(T[] nodes, ...) {
2     /** Sort weights for the nodes array */
3     TreeMap<...> weightedTree = new TreeMap<>();
4     ...
5     // Sort nodes which have the same weight.
6     for (List<T> l : weightedTree.values()) {
7       Collections.shuffle(l);
8       for (T n : l) nodes[idx++] = n;
9     }
10 +    Preconditions.checkState(idx != activeLen,
11 +        "Sorted the wrong number of nodes!");
12 }
```

Figure 4. The core code snippet of HDFS-13279. Lines 10-11 are the fixed code.

Imbalance failures can lead to severe consequences in DFSes. One such example is an imbalance failure in HDFS [33], where incorrect data distribution calculation led to service unavailability. Figure 3 illustrates the seven key steps to trigger this bug, and Figure 4 presents the core code snippet of the bug. In an HDFS cluster, the NameNode manages metadata, while the DataNodes handle the actual storage of data. After mounting a new volume and receiving data storage requests, the Load Balancer first calculates load changes and updates the storage distribution accordingly. HDFS employs a 'clusterMap' to record connected DataNodes and a 'weightTree' to sort storage load. If the tree becomes imbalanced due to a volume change, a data migration process is triggered. However, if a DataNode happens to go offline, this imbalance failure is triggered. The disconnected DataNode's status isn't promptly updated in the 'clusterMap', resulting in an erroneous perception of the DataNode as being still active. Consequently, the migrated data calculation is incorrect, creating 'hotspots' (where the data of some nodes is not migrated out, but still retained) and blocking new data stored to these hotspot DataNodes. This bug leads to service hang-ups, affecting HDFS availability. It is fixed by adding a timely state checker (lines 10-11). If $node[idx]$ is not active, then the code throws an exception and recalculates the data migration to be performed.

3.4 Limitation of Existing Methods

Currently, several state-of-the-art testing tools for complex systems, such as distributed systems and file systems, incorporate test case generation using both client requests and configuration changes. However, these tools overlook the potential execution dependencies between the two input spaces, which hampers their effectiveness in triggering imbalance failures in DFSes. Existing methods can be categorized into three main types:

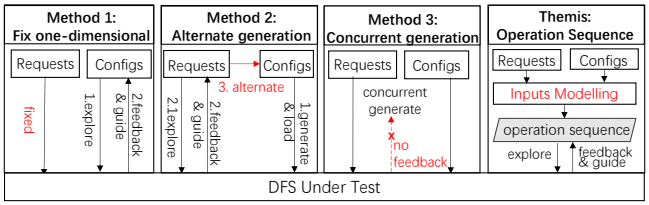

Figure 5. Methods to explore two types of input spaces are discussed. Detailed experimental comparisons between these methods are provided in Section 6.2.

Method 1: Fix one input. One straightforward approach is to fix one type of input space and explore the other. This method is commonly used by most testing tools [2, 18, 19, 45]. Figure 5 illustrates the testing process. Fault injection tools, such as CrashFuzz [19] and Mallory [45], utilize runtime feedback fuzzing to guide the generation of cluster configurations (e.g., killing node, adding node) while fixing the client request workloads to test the fault-tolerant mechanisms of distributed systems. Workload generation tools like SmallFile [2] and Filebench [18] excel in generating file-related operation workloads with fixed configuration settings. However, this approach fails to explore the combinations of request and configuration inputs, leading to ineffective detection of imbalance failures in DFSes.

Method 2: Alternate generation. Similar to file system testing tools like Janus [62], Hydra [38], and Falcon [64], the alternate generation strategy involves three key steps: (1) it begins by randomly generating a system configuration, which the DFS under test loads and executes; (2) it employs a coverage-guided fuzzing process to explore the client request input space; (3) once the exploration of the request input space is complete (indicated by test coverage converging, with no new coverage growth for an extended period), it generates a new random configuration and moves to the next iteration (repeating steps 1 and 2). Although this method does explore some combinations of the two input

spaces, it explores each input separately. As a result, it overlooks some potential execution dependencies where the two types of inputs are frequently combined and executed over a short duration, e.g., the triggering steps 'data_new, volume_remove, data_remove, node_remove' in the motivation example, which limits its ability to detect deep bugs.

Method 3: Concurrent generation. Another potential solution is to concurrently generate two types of inputs. As depicted in Figure 5, it simultaneously conducts stress testing with a high volume of client request workloads while generating numerous system configuration test inputs in parallel. However, this method struggles to effectively utilize runtime feedback for optimizing the input spaces. Since the request and configuration inputs are generated concurrently and independently, it becomes challenging to identify which input type changes trigger the current runtime status. Consequently, this method relies on a random search across the expansive input spaces, making it inefficient.

The Key insight of Themis: Unlike existing testing tools, we represent both changes to the system configuration and user requests as sequences of operations instead of considering them as two independent input spaces. Building on our Finding 6, which shows that load imbalances in DFSes result from the accumulation of minor variances, Themis employs load variance-guided fuzzing to effectively explore the execution space of operation sequences and detect imbalance failures hidden in DFSes.

4 Themis Design

Design goal: A practical imbalance failure detection framework should have the following properties.

- *General:* Themis is designed to find imbalance failures for most practical distributed file systems, from Clustered File Systems, e.g., HDFS [55], to Network File Systems (NFS), e.g., LeoFS [41], and from Distributed Block Storage, e.g., GlusterFS [20], to Distributed Object Storage, e.g., CephFS [59]. The tool can be deployed to different kinds of DFSes with minor adjustments.
- *Non-intrusive:* For most DFSes [31, 49, 52], neither can Themis directly modify their source code nor alter their software stacks. Therefore, Themis can only rely on their runtime performance status (e.g., CPU/network usage, storage distribution, etc.) for detection.
- *Efficient:* Themis is able to frequently exercise the load balancing logic and effectively detect imbalance failures in real-world DFSes within 24 hours.
- *Accurate:* Themis is designed to have satisfactory precision and recall to avoid reporting false positives.

4.1 Themis Workflow

Figure 6 illustrates the workflow of Themis, consisting of two key components: a Test Case Generator for synthesizing high-quality test cases and an Imbalance Detector for identifying imbalance failures. (1) Themis first constructs a test case input model that describes both client requests and system configurations. (2) Based on the model, Themis converts both request and configuration inputs into an operation sequence *opSeq*. (3) Test Case Generator creates initial test cases and stores them in the seeds pool [44], a collection of test cases used to generate variations through mutation. (4) Themis then generates new test cases by selecting and mutating existing ones from the seeds pool. (5) The DFS under test executes the test cases. (6) The Imbalance Detector monitors and collects the runtime load data (e.g., CPU/IO usage, storage distribution, etc.) from the DFS. (7) The load variance between distributed nodes is calculated in real-time. (8) The Load Variance Model is updated accordingly. Meanwhile, Imbalance Detector identifies the imbalanced states and reports the imbalance failures once they are detected. (9) Test cases contributing to new imbalance failures or larger load variance are prioritized to the seeds pool for guiding subsequent test case generation. Themis proceeds to the next testing iteration (from step 4 to step 9, the load imbalance gradually accumulates across iterations) until the imbalance exceeds a predefined threshold. At this point, Themis identifies that the DFS has entered a failure state, resets the DFS to its initial state and restarts the testing process.

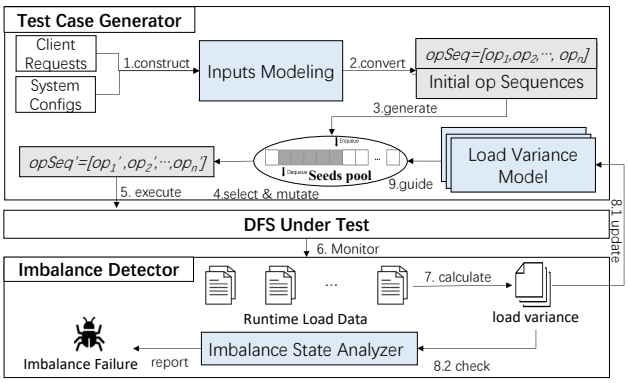

Figure 6. The workflow of Themis. It includes two main components: (1) Test Case Generator for generating high-quality test cases. (2) Imbalance Detector for identifying imbalance failures.

4.2 Test Case Generator

Both client requests and system configuration impact the load on DFS, triggering its load balancing mechanism. However, the search space for combinations of these two types of inputs is vast. To effectively explore this space, Themis first models all load-related client requests and system configurations, transforming them into an operation sequence *opSeq*, and then Themis employs a load variance-guided fuzzer to thoroughly explore the sequence space and effectively detect imbalance failures hidden in the deep logic.

Inputs Modeling: Themis uses a test case specification, as shown in Figure 7, to model all load-related operations.

```
testcase: operation+; //operation sequence opSeq
operation: opt opd+;  //opt:operator opd:operands
opt: file_op | node_op | volume_op ;
file_op:    'create'    | 'delete'  | 'append'  | 'overwrite' | 'open'
          | 'truncate-overwrite'  | 'mkdir'   | 'rmdir'     | 'rename' ;
node_op:    'add_MN'        | 'remove_MN'  |'add_storage'  | 'remove_storage';
volume_op:  'add_volume' | 'remove_volume' | 'expand_volume' | 'reduce_volume';
opd:        fileName | nodeId | size;
```

Figure 7. The test case specification of Themis that models both request and configuration inputs in DFSes.

Specifically, each test case is an operation sequence that contains at least one *operation*. Each *operation* consists of an operator and at least one operand. Operators can be divided into three main categories: *file_op* is designed to describe the client requests inputs, e.g., file create, append, delete, etc. *node_op* and *volume_op* are designed to model the system configuration inputs, e.g., node add/delete and volume expansion/reduction. Operation '*add_MN* $node_a$' means adding a metadata management node, $node_a$ and '*remove_storage* $node_b$' represents removing a storage node, $node_b$. The number and contents of operands *opd* are determined by the operator *opt*. For example, if the *opt* is *create*, then there should be at least two operands, opd_1 is the 'filename' and opd_2 is the size of the newly created file.

Initial OpSeq Generation: Before starting the testing phase, Themis needs to create initial test cases to drive the testing process. Themis first determines the maximum length, max_n, of the operation sequence *opSeq*. Guided by our finding 5 that the imbalance failures require no more than 8 triggering steps, we set max_n to 8. Then, Themis randomly generates *opSeq* with lengths varying from 1 to max_n. For each operation in *opSeq*, the operator, *opt*, is generated randomly with an equal probability of $\frac{1}{t}$, where t, the number of all distinct load-related operations, is 17 in our model. The operand, *opd*, is instantiated according to its category.

- Category FileName: Themis uses a file tree $Tree_{files}$ to manage and record all file names and their storage topological relationships within the DFS. When instantiating a file name, Themis either selects an existing FileName using a uniformly random distribution from this tree or creates a new FileName and adds it to the tree.
- Category NodeId: Themis uses two lists, $list_{MN}$, and $list_S$, to keep track of all management nodes and data storage nodes in the DFS, respectively. When instantiating the 'nodeId', a node is randomly selected from these lists according to the specific *opt*. For instance, if *opt* is *add_MN*, the nodeId is randomly chosen from $list_{MN}$. Similarly, if *opt* is *add_storage*, the nodeId is selected from $list_S$.
- Category Size: This category is used to specify the size of the data being manipulated. Themis keeps track of the remaining storage capacity, $free_{space}$, of the DFS. To more effectively test the load balancing logic, Themis creates boundary scenarios of the data size. For example, when instantiating a "create filename size" *operation*, the file size is randomly assigned a value between 0 and $free_{space}$.

Node Load Variance Model: In most DFSes, the goal of the load balancing mechanism is to achieve a balanced load distribution among nodes as possible, rather than maintaining constant uniformity. Consequently, transient and minor load differences among nodes are considered normal and acceptable. However, a core insight of Themis is that the ultimate state of load imbalance in a DFS is caused by the continuous accumulation of many intermediate states of minor load variances. Therefore, to efficiently reach the deep code logic and trigger the imbalance failures efficiently, Themis introduces the load variance model to guide the test case generation and ensure the nodes in the system experience as much load variation as possible.

Computaion Load Data

$Core_i$: the average number of CPU Cores used by $node_i$ during test case execution
Cpu_a: the usage rate of the ath CPU core

Network Load Data

$Request_i$: the average number of requests handled by $node_i$ during test case execution
$Read_i$: the network I/O read usage of $node_i$
$Write_i$: the network I/O write usage of $node_i$

Storage Load Data

$Storage_i$: the average size of data stored in $node_i$ during test case execution

Load Variance

$$Variance_{ij} = \left|\sum_{a=1}^{Core_i} Cpu_a - \sum_{b=1}^{Core_j} Cpu_b\right| + |Request_i - Request_j| + |Read_i - Read_j| + |Write_i - Write_j| + |Storage_i - Storage_j|$$

Figure 8. Definition of the Load Variance Model, which includes four parts: Computation Load Data, Network Load Data, Storage Load Data, and their Load Variance.

Figure 8 describes the load variance model of nodes in DFSes. The node's load data can be divided into three main types. Computation load data indicates the core computation distribution of each management node, including $Core_i$, the number of CPU Cores used by $node_i$, and the utilization of each CPU core Cpu_a. Network load data presents the key real-time network IO usage information of each distributed node. Note that $Request_i$ indicates the number of requests handled by $node_i$; $Read_i$ means the network read IO usage (i.e., the number of input operations) of $node_i$ and $Write_i$ presents the network write IO usage (i.e., the number of output operations) of $node_i$. The storage information $Storage_i$ indicates the size of data stored in $node_i$. During the execution of each test case, the load information for each node is calculated. The load variance between two nodes is quantified by the sum of their respective load differences. Specifically, the computation load difference between $node_i$ and $node_j$ is calculated as $|\sum_{a=1}^{core_i} Cpu_a - \sum_{b=1}^{core_j} Cpu_b|$. The difference in client requests processed by $node_i$ and $node_j$ is given by $|Request_i - Request_j|$. The difference in read I/O usage between $node_i$ and $node_j$ is calculated as $|Read_i - Read_j|$, and similarly, $|Write_i - Write_j|$ quantifies the difference in write I/O usage. The storage load difference between $node_i$ and $node_j$ is measured as $|Storage_i - Storage_j|$. However, the weights of three variance types may have varying impacts

on testing performance, and we will discuss it in detail in Section 7.

Load variance-guided Fuzzing: Themis employs load variance guided fuzzing to explore the input space of test cases, i.e., operation sequences. In each fuzzing iteration, Themis first dequeues a sequence *opSeq* from the seed pool and mutates it to *opSeq′* by the *OpSeq Mutation* strategy, as explained below. Then the operations in *opSeq′* are sent to the DFS to be executed, and the load states of distributed nodes are updated and analyzed in real-time. Based on the load variance model, variances among distributed nodes are calculated. In the meanwhile, our imbalance detector analyzes the variances to determine if it indicates an imbalanced state. The detailed imbalance failure detection process will be introduced in Section 4.3. If the variance becomes larger or any new imbalance failures are found, then the new test case *opSeq′* will be regarded as an interesting seed and stored in the seed pool to guide the next fuzzing iteration. In this way, Themis constantly generates high-quality operation sequences as test cases, aiming to maximize the load imbalance across distributed nodes.

OpSeq Mutation: Similar to the mutation process used by the state-of-the-art tool AFL [21], Themis proposes three types of mutation operations: (1) *opt* replace; (2) *opt* delete; and (3) *opt* insert. For a given operation sequence *opSeq*, Themis first randomly selects a set of positions $P = \{p_a, p_b, ..., p_k\}$ within *opSeq*, where $k \leq length(opSeq)$. For each selected position p_i, Themis performs the corresponding mutation operation based on the type specified. (1) Replace Mutation, Themis replaces the p_ith *opt* in *opts* with a new randomly created *opt′*. (2) Delete Mutation, Themis removes the p_ith *opt* from the sequence. (3) Insert Mutation: Themis generates a new *opt* based on our model and inserts it at the p_ith position. Subsequent to the *opt* mutations, the operands for the new or modified operations are instantiated in the same manner as described in *Initial opSeq Generation*. Note that for each mutated *OpSeq*, we will scan all its *opt*s and check whether an *opt* references a file or node that no longer exists. If such a reference is found, the *opt* will be updated by replacing its FileName or NodeId with a random one from $Tree_{files}$, $list_{MN}$, or $list_S$.

4.3 Imbalance Detector

Inspired by finding 3, the Imbalance Detector is designed to monitor the runtime load states of nodes in the DFS and identify any unreasonable load imbalances.

Based on the imbalance definition in Section 2.2, we propose the Imbalance Detector, as shown in Figure 9. After executing the test cases generated by Themis, the States Monitor collects runtime load data and calculates the Load Variance Model (LVM_{ij}) for each pair of nodes, $node_i$ and $node_j$, within the DFS. Three anomaly detectors are then employed to separately assess computation load, network load, and storage load variances. These detectors determine

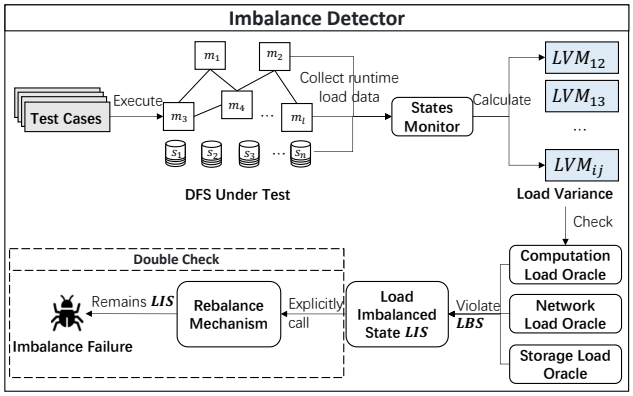

Figure 9. Imbalance Detector in Themis for monitoring the load state and identifying imbalance failures in real-time.

whether the DFS has entered a Load Imbalanced State by verifying if the maximum load among the nodes exceeds the average node load multiplied by the variance threshold, *t*. If an imbalanced state is detected, a candidate imbalance failure is identified. To confirm its validity and minimize false positives, a double-check process is performed.

Double Check: In practice, different DFSes use various load balancing mechanisms. For example, CephFS calculates and balances loads in real time, whereas GlusterFS uses periodic timing tasks for load balancing. Consequently, the time required to complete test cases, activate the load balance mechanism, and achieve a balanced load state differs across DFSes. This variation makes it hard for Themis to precisely determine when detectors should check the load state to identify imbalances without producing false positives. The appropriate checking time varies significantly depending on the specific DFS implementation and factors in its distributed runtime environment, such as latency, bandwidth, etc. Fortunately, most DFSes provide rebalance APIs that enable users to directly trigger and execute their load balance mechanisms. To filter out false positives, Themis incorporates a double-checking process. Specifically, when a candidate imbalance failure is identified, Themis explicitly calls the rebalance API. Following this, Themis invokes the 'rebalance state' API to confirm whether the rebalance operation has been completed. Once the API returns 'rebalance done', Themis immediately re-executes the test case and checks the current load state. If the DFS still remains in imbalance state, the candidate failure is confirmed as a true positive imbalance failure.

In Themis, the variance threshold value *t* is used to determine the extent of load difference between two nodes that constitutes an imbalance. This threshold is crucial for the accuracy of imbalance detection. Setting the value *t* too low may result in numerous false positives because, in practical DFS implementations, the load distribution among nodes is typically relatively balanced but not equal, and minor imbalances are considered normal and acceptable. Conversely,

setting the value t too high could compromise the effectiveness of imbalance detection, as it may miss some potential imbalance failures. How to find a optimal t will be thoroughly explored and discussed in detail in Section 6.4.

5 Implementation

We implemented Themis on four widely-used DFSes, using their latest versions: HDFS v3.4, CephFS v18.0.0, GlusterFS v12.0, and LeoFS v1.4.4. The reasons for choosing them are listed below:

DFS Popularity: Hadoop Distributed File System (HDFS) [55], a linchpin in Apache Hadoop, is widely embraced by organizations like Facebook [5] and Yahoo [6] for its scalability and fault tolerance. GlusterFS [20], tailored for high-performance computing, is lauded by major industry users such as Red Hat [50] and Cisco [35]. CephFS [59], integral to the Ceph storage platform, is selected by many entities like CERN [40] and DigitalOcean [11]. LeoFS [41], prioritizing high availability, favored by enterprises like Hyperscalers [30].

DFS Diversity: These distributed systems come from different organizations with different implemented languages. HDFS is developed by Apache Software Foundation in Java language. GlusterFS is developed by Red Hat and implemented in C++. LeoFS is developed by Rakuten in Erlang. CephFS is developed by the Ceph community in C++ language. Implementation and evaluation of these DFSes can demonstrate that Themis is a cross-platform and language-independent testing framework with high generality.

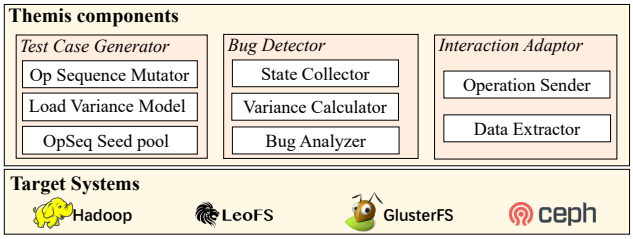

Figure 10. Core components of Themis implementation, contain three key parts: Test Case Generator, Imbalance Detector, and Interaction Adaptor.

Figure 10 presents the components of Themis, which can be divided into three main parts. The first part is the Test Case Generator which is implemented for synthesizing high-quality test cases that model both request and configuration inputs, and diverge the load of DFS as much as possible. The second part is the Imbalance Detector for monitoring the load variance of distributed nodes and identifying load imbalance failures. The third part is the Interaction Adaptor, which is designed to interact with target systems, including sending specific load-related operations (e.g., create/remove files, expand/reduce volume, etc.) and collecting load data of the DFS under test. The first and second parts are independent of the DFS under test. Only the third part requires minor modifications when adapting to a new DFS. The rest of the section describes notable implementation details.

Adaption to New Distributed File Systems: The effort of adapting Themis to other DFSes could be small. Modules in Themis are well-encapsulated and loosely coupled. Hence, when adapting Themis to a new DFS under test, developers only need to implement two interfaces related to a specific DFS. (1) The first interface is 'operation.send()', to send operations to DFS for execution. Specifically, for file operations, since most of the DFSes support the Filesystem in Userspace (FUSE) [37] that allows users to access and interact with distributed file systems seamlessly, enabling the mounting of remote file systems as if they were local, all file operations produced by Themis conform to Fuse's interface specifications, so this type of operations does not require any adaptation costs. For the Nodes and Volume operations, we need to implement an adaptor that converts the operations in Themis to commands in the target DFS (e.g., convert operation *'remove_volume gluster1'* in Themis to operation *'gluster volume remove-brick Themis-Test gluster1:brick1 start'* in GlusterFS). (2) The second interface is 'LoadMonitor()', which is responsible for monitoring key load data in each distributed node. For instance, to gather detailed storage information, e.g., used space, free space, etc., on the disks mounted by the DFS, we use the system command *'df | grep Disk_MountedbyThemisTest'* to retrieve this data in real-time.

Imbalance Reproduce, Diagnose and De-duplicate: When Themis detects an imbalance state, it automatically sorts and records all operations performed by each node, organizing them by timestamp in a reproduction log. This step of operations logging is automatic. We then provide this reproduction log, along with the comprehensive DFS execution logs (LOG.LEVEL=ALL), to the developers. Using these operation sequences, we attempt to replay the operations according to chronological order to reproduce the imbalance-triggering process and manually analyze the root cause. Once the imbalance is successfully reproduced and its root cause identified, developers confirm the failure and implement the necessary fixes. If two Imbalance Failures share the same root cause, we consider them duplicates and remove the one with the longer operation sequence. This step of analyzing imbalance failures is manual. Thanks to our testing environment setup, where imbalance detection is conducted using virtual machines on a single computer, we can effectively avoid the impact of other input factors such as network delays and hardware faults on the test results. Consequently, imbalance failures can be reliably reproduced based on the reproduction log.

6 Evaluation

To evaluate the effectiveness of Themis, we compared it with three state-of-the-art methods widely used in distributed system testing: Fix one input space, Alternate generation,

Table 2. 10 new imbalance failures were detected by the tools within 24 hours. Themis found all 10 imbalance failures, including 4 in GlusterFS, 3 in LeoFS, 1 in CephFS, and 2 in HDFS.

#	Platform	Failure Type	The Root Cause Analysis	Identifier
1	GlusterFS	Imbalanced Storage	load imbalance due to mistakenly removing plenty of file data in dht.rebalancer, causing serious data loss in GlustreFS.	Bug#S24387
2	GlusterFS	Imbalanced Storage	Imbalanced storage distribution after mistakenly handling plenty of file operations with large size differences in gf.handler.	Bug#S24389
3	GlusterFS	Crash	Some nodes in the network crash down after frequently executing load rebalance commands due to a null-pointer hashID.	Bug#S25081
4	GlusterFS	Imbalanced CPU	Imbalanced computation load caused by wrong assignment in gf_self_healing after nodes change and surge in client requests.	Bug#S25088
5	LeoFS	Imbalanced Storage	Storage distributes unevenly due to wrong rebalance_list read in leofs.cluster after constant file resizing and volume changing.	Bug#S231116
6	LeoFS	Imbalanced Storage	Some nodes become 'hotspots' caused by incorrect data sync in leofs.migration after nodes enter and exit frequently.	Bug#S231117
7	LeoFS	Imbalanced Network	Requests distrusted imbalance due to wrong rebalance measuring between two LeoGateways when two nodes happen to exit.	Bug#S231137
8	CephFS	Imbalanced Storage	Imbalanced storage where some storage devices are full while others only occupy 65% caused by balancing IO hangs in replicas.	Bug#63890
9	HDFS	Imbalanced Storage	Some disks become 'hotspots' due to Inode conflicts in balancing when executing many file operations within nodes scaling.	Bug#20240111
10	HDFS	Imbalanced Network	NameNodes traffic jams due to blocks in newly generated files in checkpointSize when some storage replicas went offline.	Bug#20240126

and Concurrent generation, on four widely used DFSes. We ran each DFS in a cluster of 10 virtual nodes isolated by Docker [46]. Each Docker container had a 2.25 GHz 6-core CPU, 16 GB of RAM, and a 480 GB SATA SSD. They were connected via a 10 Gbps network bandwidth. They ran Ubuntu 20.04.2 with Linux kernel version 4.4.0. All containers were hosted on a physical machine, a 64-bit system with 128 CPU cores (AMD EPYC 7742 64-Core Processor), and 512 GB main memory. All the experiments were conducted multiple times with the same environment setups, and the average values are used in this paper. We designed experiments to address the following research questions.

- **RQ1:** Is Themis effective in finding imbalance failures in real-world DFSes?
- **RQ2:** Can Themis cover more code of DFSes compared with state-of-the-art methods?
- **RQ3:** Does the load variance model effectively improve testing performance?
- **RQ4:** How does the variance threshold t influence the accuracy and false positives of Themis?

6.1 Imbalance failures in real-world DFSes.

We applied Themis to all four DFSes under test for imbalance failure detection evaluation. Distributed system testing tools, e.g., CrashFuzz [19] explore system fault inputs with fixed client request inputs. But they only support injecting configuration faults of killing and adding nodes. To enable a fair comparison, we modified Themis to align the strategy with CrashFuzz, which we call Fix_{req}. Specifically, we first generate a set of client requests as the fixed inputs. Then, we replace Themis' load variance-guided fuzzing with CrashFuzz's coverage-guided strategy to explore the system configuration inputs. DFS testing tools, e.g., SmallFile [2], generate plenty of file operations with fixed system configuration, we ran it on the same DFSes, labeled as Fix_{conf}. File system fuzzers, e.g., Janus [62] alternately explore file operation inputs and file image inputs. To facilitate comparison, we adapted Janus's alternate exploration method in Themis but replaced its file image inputs with our system configuration inputs, denoted as *Alternate*. Additionally, we also concurrently generate client requests and system configurations, which we call *Concurrent*. For comparison, we ran all the tools on the same DFSes using the same experimental setup. Considering that all existing methods lack an effective detector for imbalance failures, we enhanced them with our imbalance detectors. Each experiment was conducted for 24 hours. In total, Themis produced 60,000+ operations, successfully detected 16 imbalance issues, with 10 being identified as distinct imbalance failures. Detailed information on these previously unknown imbalance failures is presented in Table 2.

As shown in Table 2, the majority (6/10) of imbalance failures caused uneven data distribution across distributed storage nodes, creating 'hotspots' that degraded overall DFS performance and even blocked the provision of certain functionalities. Two imbalance failures (#7 and #10) led to uneven handling of network requests by the metadata management nodes, causing numerous client requests to be suspended. One imbalance failure (#4) resulted in an unbalanced CPU usage load on distributed nodes, leaving some nodes overloaded all the time, unable to provide function properly, and they were unable to recover themselves. One imbalance failure (#3) caused storage nodes to crash, and the distributed nodes could not be recovered automatically. Some of the imbalance failures can lead to serious consequences. Take failures #3, #6, and #10 for example, attackers may deliberately crash or hang specific target nodes, disrupting their ability to handle requests or impeding the data synchronization process through the execution of specific load-related operations. This malicious activity has the potential to directly result in the loss of critical data, cause the outage of essential functions in cloud services, and consequently lead to significant economic losses.

Table 3. Imbalance failures found by Themis and other state-of-the-art methods. Other methods detect no more than 4 failures, while Themis detects all 10 imbalance failures.

Method	Themis	Fix_{req}	Fix_{conf}	*Alternate*	*Concurrent*
Number	10	1	2	3	4
Bugs ID#	#1 - #10	#5	#2, #9	#2, #5, #9	#2, #3, #5, #9

Comparison with existing methods: In our 24-hour experiments, Fix_{req} only found 1 imbalance failure (#9). Fix_{conf} detected two failures (#2, #9). *Alternate* and *Concurrent* successfully detected 3 failures (#2, #5, #9) and 4 failures (#2, #3, #5, #9) respectively. However, the remaining 6 imbalance failures were not found by them because these imbalance failures are hidden in the deep logic. To trigger them, several operations (at least 6 steps) involving both types of inputs, i.e., client requests and system configuration changes, need to be executed first. Unfortunately, existing methods lack efficient strategies for exploring two input spaces, ignoring their execution dependencies and thereby missing these imbalance failures. In contrast, Themis models both request and configuration inputs into a sequence and employs load variance-guided fuzzing to effectively explore the execution space of the operation sequences. This approach allows Themis to successfully exercise the load balance logic in the deep path and detect all 10 imbalance failures, proving the effectiveness of Themis in detecting imbalance failures in real-world DFSes, which adequately answers **RQ1**. Compared with other state-of-the-art testing techniques, Themis found all the imbalance failures that other methods found.

6.1.1 Case Study. We will now use one case to illustrate how imbalance failures detected by Themis impact the overall distributed file system and how Themis identifies this failure. This case corresponds to failure #1 listed in Table 2. This is an imbalance failure in GlusterFS, discovered in version v12.0. The failure was caused by incorrect file deletion within the rebalance code implementation, which potentially led to arbitrary data loss, compromising the system's availability and reliability. During our testing process, this error was frequently triggered, causing significant data loss and leading to an imbalanced storage distribution, which was eventually detected by Themis. The code snippet in Figure 11 describes details of this imbalance failure and its fixed code.

```
1  static void rebalance(...){
2      ...
3      for(auto file: queue) gf_migrate_file(file);
4  }
5  static int gf_migrate_file(...){
6      if (cached_list.contains(hashed_id)) {
7  -       if (is_linkfile == 1) {
8  -           i = rebal_entry->local_index;
9  -           ret = syncop_unlink(local_subvols[i]...);}
10 +       i = rebal_entry->local_index;
11 +       link_id = local_subvols[i];
12 +       if (is_linkfile == 1 && hashed_id != link_id) {
13             ret = syncop_unlink(linkfile_id, ...);
14             gf_msg_debug("Unlink linkfile");
15         }
16 }
```

Figure 11. An imbalance failure due to incorrect file deletion in data migration within the dht-rebalance.c of the GlusterFS.

Root Cause: In GlusterFS, when the storage load becomes imbalanced, the 'rebalance()' function is invoked to redistribute the storage load, initiating the data migration process. In the scenario where a datafile *fd* has recently undergone migration and its hash_id remains in the cache, if the rebalancing mechanism is subsequently triggered and executed again and *fd*'s linkfile (hard link or soft link) happens to require migration, GlusterFS first checks whether its hashed_id is in cached_list. Given that the linkfile shares the same hashed_id as its original datafile *fd*, this linkfile is erroneously deleted, as shown in lines 6-9. Consequently, this sequence triggers the imbalance failure, leading to the erroneous removal of linkfiles from GlusterFS. The developer has fixed this imbalance failure by adding an extra linkfile_id check, as shown in lines 10-12. This imbalance failure poses a severe security threat to GlusterFS, enabling the deletion of an arbitrary number of linkfiles stored in distributed storage nodes.

In our experiments, this imbalance failure was only detected by Themis. It is difficult to detect due to the complexity of the required long triggering sequence: 'create *fd* -> data changes (via file-related operation requests) -> load rebalance -> migration filedata *fd* -> load changes (by either requests or configs) -> load rebalance again -> migration *fd*'s linkdata'. Such a deep imbalance failure can only be activated under specific execution dependencies where the execution interval between two rebalancing processes is sufficiently short, and both the file data and its linked data are migrated in the correct execution sequence. Other methods often fail to detect this failure because they cannot effectively track such complex execution dependencies. Fortunately, Themis employs load variance-guided fuzzing that effectively explores the input execution dependencies by consistently expanding the load variances among nodes, thereby frequently activating the load rebalancing mechanism. As a result, Themis is more likely to manipulate the datafile and linkfile within the same cache_list, successfully triggering and identifying this imbalance failure.

6.1.2 Historical imbalance Evaluation. We also evaluated Themis and other tools against 53 historical imbalance failures that we had previously analyzed. Specifically, we configured the four target DFSes with their historical versions and ran each tool to test each system over 24 hours. Themis successfully detected 48 of the 53 historical imbalance failures, Fix_{req}, Fix_{conf}, *Alternate* and *Concurrent* only detected 9, 11, 16, 21, and 23 imbalance failures, respectively, as shown in the table 4. Compared to these methods, Themis detects 109% - 433% more historical imbalances.

Of the five imbalance failures that Themis could not detect, two (CephFS #41935 [58] and HDFS #4261 [16]) only occur on the Windows OS, where Themis has not been implemented. The remaining three failures (CephFS #55568 [12], GlusterFS #1699 [4] and HDFS #11741 [8]) not only require specific

Table 4. Historical imbalance failures reproduced by Themis and other tools.

Tools	HDFS	CephFS	GlusterFS	LeoFS	Total
Themis	16/18	14/16	11/12	7/7	48/53
Fix_{req}	3/18	3/16	2/12	1/7	9/53
Fix_{conf}	4/18	3/16	2/12	2/7	11/53
Alternate	6/18	4/16	4/12	2/7	16/53
Concurrent	8/18	6/16	5/12	2/7	21/53

request and configuration inputs but also depend on particular hardware failures, such as compatibility issues between HDDs and SSDs or failures in encryption hardware. These hardware-related failures are not currently addressed by Themis. These five imbalance failures fall outside the scope of this paper and are currently not supported by Themis.

6.2 Effectiveness on Code Coverage

Since code coverage is fundamental to error detection, achieving higher coverage increases the likelihood of uncovering errors. We also evaluated Themis's ability to achieve code coverage in the DFS under test. We established a 10-node network for each target DFS and compared Themis with other state-of-the-art methods in the same experimental setup. According to the empirical study of abstract fuzzing [51], code, block, and branch coverage are highly correlated. Therefore, we just collected the branch coverage for each tool in 24 hours as the evaluation metric. The statistics are shown in Table 5. For CephFS and GlusterFS, which are written in C++, we used gcov [15] for coverage collection. Since LeoFS is implemented in Erlang, we used ExIntegration [66] to collect coverage. For HDFS implemented in Java, we used JaCoCo [32]. In conclusion, Themis always outperforms other methods on all four DFSes under test. Compared to methods Fix_{req}, Fix_{conf}, *Alternate* and *Concurrent*, Themis covers 18%, 21%, 13%, and 10% more code branches on average. The statistics adequately answer **RQ2**.

Table 5. Branch coverage on four target DFSes in 24 hours. Themis always outperforms other methods on all DFSes.

Method	Fix_{req}	Fix_{conf}	*Alternate*	*Concurrent*	Themis
HDFS	34,065	32,913	35,296	36,448	39,872
GlusterFS	42,163	41,072	43,815	44,597	49,320
LeoFS	9,413	9,204	10,026	10,318	11,529
CephFS	55,926	54,212	57,042	58,206	64,052

Compared to Fix_{req} and Fix_{conf}, Themis consistently outperforms them on all four DFSes, covering 18% and 21% more branches on average. The primary reason is that Fix_{req} and Fix_{conf} are limited to exploring one type of input space (either client requests or system configurations), thus missing the code logic that involves interactions between these two type of inputs. Themis also achieves better performance than *Alternate* on all targets, covering an additional 13% branches. This is because alternating exploration of two input spaces neglects many potential operation combinations, leading to less efficient testing. Even when compared to *Concurrent*, which theoretically can explore any combination of request and configuration inputs, Themis always performs better and covers 15,204 more branches. The main advantage of Themis over *Concurrent* is that while *Concurrent* explores input combinations randomly, Themis leverages runtime load variance feedback to guide the generation of high-quality test cases, resulting in more effective testing.

To track the trends of coverage growth over time, we recorded the branch coverage every minute over 24 hours, as shown in Figure 12. The data shows that Themis's branch coverage grows significantly during the first 4 hours on all four target DFSes. After approximately 12 hours, the coverage of Themis gradually converges (only less than 1% branch coverage improvement is observed). In comparison, the code coverage for Fix_{req}, Fix_{conf}, *Alternate*, and *Concurrent* experiences a sharp increase within the first 60-180 minutes of testing. However, beyond this initial burst, the test case inputs from these methods struggle to achieve additional coverage compared to their early performance. Throughout the testing period, Themis consistently outperforms the other four methods in terms of code coverage across all target DFSes, thanks to its load variance-guided test case generation for exploring more code logic.

6.3 Effectiveness of Load Variances Model

To evaluate the effectiveness of the Load Variance Model, we conducted an experiment comparing Themis with $Themis^{-}$, a version of Themis that disables the load variance model and generates operation sequences randomly. We collected the branch coverage and the number of imbalance failures detected by both versions in 24 hours on all four DFSes.

As shown in Table 6, with the help of the load variance model, Themis can detect 10 imbalance failures in 24 hours, while $Themis^{-}$ only detects 5 of them. Specifically, compared with $Themis^{-}$, Themis always achieved higher branch coverage on all four target DFSes. In total, Themis covers 16,249 more code branches, achieving an improvement of 11% branch coverage. Thus, we can conclude that the load variance model helps achieve better performance on both code coverage and imbalance detection. It significantly improves testing performance, which adequately answers **RQ3**.

6.4 Accuracy of Themis

To evaluate the impact of the variance threshold value t on accuracy and false positives, we conducted experiments running Themis with various t value setups, ranging from 5% to 35% of the average load across distributed nodes. We collected all the imbalance failures reported by Themis over a 24-hour period across all four target DFSes. Subsequently,

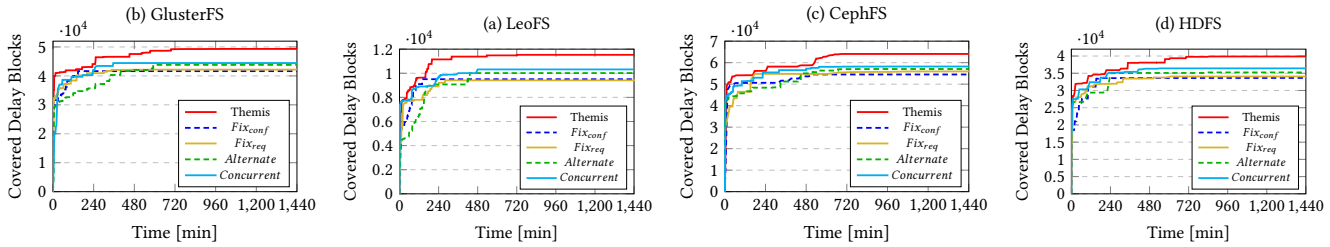

Figure 12. Coverage trends evaluated for Themis, Fix_{conf}, Fix_{req}, *Alternate*, and *Concurrent*. Compared with them, Themis with the load variance model shows better branch coverage all the time on all the target DFSes.

Table 6. Comparison of $Themis^{-}$ and Themis on four DFSes under test in 24 hours. Themis with load variance model detects 50% more failures and covers 11% more branches.

	Number of Failures		Code Coverage	
	$Themis^{-}$	Themis	$Themis^{-}$	Themis
HDFS	1	2	36,401	39,872
GlusterFS	2	4	44,586	49,320
LeoFS	2	3	10,207	11,529
CephFS	0	1	57,330	64,052
Improvement	-	+50%	-	+11%

we manually analyzed the false positives among the reported imbalance failures.

Table 7. The False Positives and True Positives of Themis on various threshold t value setups.

Threshold t	5%	10%	15%	20%	25%	30%	35%
False Positives	11	7	3	1	0	0	0
True Positives	10	10	10	10	10	9	8

As shown in Table 7, the number of false positives reported by Themis decreases significantly as the threshold t value increases, dropping from 11 to 0. However, once the value of t exceeds 25%, Themis begins to miss some true positives, with the number increasing from 0 to 2. These findings indicate that while a higher t value effectively reduces false positives, setting it too high may result in missed true positives, which adequately answers **RQ4**. Based on our experimental results, a threshold t of 25% represents an optimal balance, eliminating all false positives across the four target DFSes without missing any true positives.

7 Discussion

More bug types support. Currently, Themis supports imbalance failure detection by calculating and checking the load variance among the distributed nodes in real time. Themis has already been adapted to four widely used DFSes and has found 10 new imbalance failures. However, there are still some other types of bugs, e.g., fail-slow [23, 43] bugs, metadata inconsistency bugs [65], etc., hidden in DFSes.

Take metadata inconsistency bugs, for example, in a DFS, metadata plays a crucial role in recording and managing file structures and data properties. Inconsistent metadata in distributed systems can lead to unrecoverable conflicts, resulting in severe consequences that significantly disrupt system operations. We can adapt Themis by checking whether the metadata information of distributed nodes is constantly consistent. However, the structure and semantics of metadata information (e.g., structures, authority, etc.) are complex and various in different DFSes, which makes it hard to propose a general anomaly detector to precisely identify metadata inconsistency bugs. For example, glusterFS uses a distributed hash table [17] for managing metadata while CephFS uses a distributed metadata server cluster based on a dynamic subtree partitioning [14]. To address it, a scalable and precise anomaly detector for metadata inconsistency bugs needs to be explored in future research.

Weighting factors of load variance. Themis represents load variance as the sum of CPU variance, network variance, and storage variance with the same weighting factor, 1/3. However, these three types of variances may have varying impacts on testing performance. Their impacts may differ across different types of distributed systems. For distributed computing systems, the CPU variance may have a higher impact, and for distributed storage systems, the storage variance is probably more important than other variance types. We did a preliminary experiment by increasing the weight of storage load variance and found that the speed of triggering storage imbalances(#1, #2, #5, #6, #8, and #9) was accelerated, as shown in Table 8. A more in-depth exploration of how to assign weights to these variances when testing different systems needs to be conducted in the future.

Table 8. The average time used by Themis to trigger imbalanced storage failures on various weight factors.

Weighing Factor of Storage Load	1/6	1/3	1/2	2/3	1/1
Average Time to Trigger Imbalances (min)	498	372	359	326	302

Sensitivity to threshold t: The effectiveness of Themis is highly sensitive to the threshold t. According to our evaluation in Section 6.4, a threshold of 25% is optimal, whereas 20% yields false positives and 30% results in false negatives.

To address this, we propose two potential alternative approaches for future development: (1) Dynamic Adjustment of threshold t: When testing a DFS with Themis, we could initiate the imbalance detector with a lower t value (e.g., 20%) and incrementally increase it upon encountering false positives. This approach allows for adaptive thresholding based on real-time testing feedback. (2) Incorporating machine learning technology: Instead of using a fixed threshold, we could train an AI model with practical load data from real-world DFSes to determine when the system enters load imbalance states.

Imbalance false positive: Currently, Themis uses load variance with a fixed threshold to identify imbalance states. However, in some special scenarios, this imbalance detector may produce false positives. For instance, if all clients are only reading a single file, the load will be concentrated on the replicas of that file, leaving other nodes idle. In this case, the detector might incorrectly identify this as an imbalance failure. However, during the initialization process, Themis randomly generates a large number of files. Additionally, during the testing process, for each *opt* in the test case, Themis randomly generates the FileName' and NodeId', preventing such rare occurrences (where all clients read a single file) from happening during our 24-hour experiment. In the future, we plan to improve our imbalance detector via dynamic threshold adjustment or AI model to address this issue, as discussed above.

.

8 Related Work

File System Testing. In the field of testing DFSes, various file load generation tools have been developed to assess and test system performance and resilience. Tools such as SmallFile [18] and Filebench [2] are proposed for simulating distributed workloads on multiple hosts, offering insights into the robustness and efficiency of DFSes. For traditional File System testing, fuzzing tools such as Syzkaller [22], and kAFL [53], are developed to generate system call sequences based on predefined grammar rules and feedback from runtime code coverage. However, these tools primarily focus on file-related operations and ignore system configuration, making them less effective for detecting imbalance failures in DFSes. Two input spaces testing tools, such as Janus [62] and Hydra [39], initially create a random file image and then apply coverage-guided fuzzing to generate system calls for this image. They alternately explore the two input spaces. However, this method fails to explore the potential execution dependencies between request and configuration inputs, leading to ineffectiveness in detecting imbalance failures.

Distributed System Testing. Some tools utilize fault injection technology to identify and address potential system failures by deliberately injecting faults or errors. Distributed system fuzzing tools, such as CrashFuzz [19] and Mallory [45], manipulate cluster faults, including killing nodes and adding nodes, similar to configuration operations included in Themis. Specifically, CrashFuzz employs coverage-guided feedback to optimize fault injection locations and expose crash recovery bugs in distributed cloud systems. Mallory applies runtime timeline-driven testing and timeline abstraction for adaptively injecting cluster faults. However, they only explore the cluster faults input generation with either fixed or random workload inputs, which is ineffective in detecting imbalance failures.

9 Conclusion

In this paper, we propose Themis, a testing framework for automatically detecting imbalance failures in DFSes. Themis first models both request and configuration inputs and converts them into an operation sequence. Then, Themis proposes a load variance-guided fuzzing to effectively explore the sequence input space and constantly generate high-quality test cases to make nodes loaded as differently as possible. We implemented and evaluated Themis on four widely used DFSes: HDFS, GlusterFS, LeoFS, and CephFS. Themis covers 10% - 21% more code compared with other state-of-the-art methods. Themis successfully detected 10 new imbalance failures. Our future work will focus on supporting more bug types and exploring better alternative approaches for fixed threshold t.

10 Acknowledgments

We would like to thank our shepherd, Martin Kleppmann, and the anonymous EuroSys reviewers for valuable feedback and input on this paper. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002).

References

[1] Mahdi S Almhanna, Tariq A Murshedi, Firas S Al-Turaihi, Rafah M Almuttairi, and Rajeev Wankar. Dynamic weight assignment with least connection approach for enhanced load balancing in distributed systems. 2023.

[2] Distributed System Analysis. Distributed metadata-intensive workload generator for POSIX-like filesystems. https://github.com/distributed-system-analysis/smallfile, 2024. Accessed at January 1, 2024.

[3] GlusterFS Balancer. https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/rebalance/, 2023.

[4] Bockeman. One brick offline with signal:11 received during rebalance healing process. https://github.com/gluster/glusterfs/issues/1699, 2020. Accessed at March 27, 2024.

[5] Dhruba Borthakur. Apache Hadoop filesystem and its usage in Facebook. https://profile.iiita.ac.in/bibhas.ghoshal/Cloud_and_Edge_Computing/hdfs_iit.pdf, 2010. Accessed at March 27, 2024.

[6] Edward Bortnikov. Yahoo's infrastructure harnesses Hadoop distributed file system (HDFS) for ultra-scalable storage). https://developer.yahoo.com/blogs/138739227316, 2016. Accessed at March 27, 2024.

[7] Chen. Force-migration.t execute failed. https://github.com/gluster/glusterfs/issues/3513, 2022. Accessed at March 27, 2024.

[8] Wei-Chiu Chuang. Long running balancer may fail due to expired DataEncryptionKey. https://issues.apache.org/jira/browse/HDFS-11741, 2017. Accessed at March 27, 2024.

[9] Hsueh-Yi Chung, Che-Wei Chang, Hung-Chang Hsiao, and Yu-Chang Chao. The load rebalancing problem in distributed file systems. In *2012 IEEE International Conference on Cluster Computing*, pages 117–125. IEEE, 2012.

[10] Loic Dachary. PG autoscaler tuning => catastrophic ceph cluster crash. https://tracker.ceph.com/issues/64333, 2021. Accessed at March 27, 2024.

[11] Anthony D'Atri. Why we chose Ceph to build block storage). https://www.digitalocean.com/blog/why-we-chose-ceph-to-build-block-storage, 2024. Accessed at March 27, 2024.

[12] Tatjana Dehler. CephPGImbalance alert inaccuracies, causing imbalanced storage load). https://tracker.ceph.com/issues/55568, 2021. Accessed at March 27, 2024.

[13] Shyam C Deshmukh and Sudarshan S Deshmukh. A survey: Load balancing for distributed file system. *International Journal of Computer Applications*, 111(5):25–29, 2015.

[14] CephFS Doc. Dynamic subtree partitioning with balancer on specific ranks. https://docs.ceph.com/en/reef/cephfs/multimds/, 2024. Accessed at January 1, 2024.

[15] Gcov documentation. A test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2024. Accessed at January 1, 2024.

[16] Junping Du. Timeouts in load-balancing process within MiniDFS-Cluster NodeGroup. https://issues.apache.org/jira/browse/HDFS-4261, 2017. Accessed at March 27, 2024.

[17] Damon Earp. *Heuristics in Distributing Data and Parity with Distributed Hash Tables*. PhD thesis, Auburn University, 2021.

[18] Filebench. File system and storage benchmark that uses a custom language to generate a large variety of workloads. https://github.com/filebench/filebench, 2024. Accessed at January 1, 2024.

[19] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. Coverage guided fault injection for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2211–2223. IEEE, 2023.

[20] Gluster. Gluster filesystem : Build your distributed storage in minutes. https://github.com/gluster/glusterfs, 2024. Accessed at January 1, 2024.

[21] Google. American fuzzy lop. https://github.com/google/AFL, 2023. Accessed at January 1, 2024.

[22] Google. syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2023. Accessed at October 23, 2023.

[23] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[24] Apache Haddop. HDFS Disk Balancer. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSDiskbalancer.html, 2024. Accessed at March 27, 2024.

[25] Azure Service Health. Final root cause analysis and improvement areas: Nov 18 Azure storage service interruption. https://azure.microsoft.com/en-us/blog/final-root-cause-analysis-and-improvement-areas-nov-18-azure-storage-service-interruption/, 2024. Accessed at January 1, 2024.

[26] Taufik Hidayat, Yasep Azzery, and Rahutomo Mahardiko. Load balancing network by using round robin algorithm: a systematic literature review. *Jurnal Online Informatika*, 4(2):85–89, 2019.

[27] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[28] Hung-Chang Hsiao, Hsueh-Yi Chung, Haiying Shen, and Yu-Chang Chao. Load rebalancing for distributed file systems in clouds. *IEEE transactions on parallel and distributed systems*, 24(5):951–962, 2012.

[29] Dan Huang, Dezhi Han, Jun Wang, Jiangling Yin, Xunchao Chen, Xuhong Zhang, Jian Zhou, and Mao Ye. Achieving load balance for parallel data access on distributed file systems. *IEEE Transactions on Computers*, 67(3):388–402, 2017.

[30] Hyperscalers. The hyperscalers LeoFS storage appliance). https://www.hyperscalers.com/LeoFS-file-system-market-proven-software-storage-hardware-x86-AI-Telecom-media-internet-POSIX-servers-Ip-networks-architecture, 2024. Accessed at March 27, 2024.

[31] Dell EMC Isilon. Create, manage and deliver next-generation digital media content. https://www.dell.com/en-hk/dt/solutions/media-entertainment.htm, 2024. Accessed at January 1, 2024.

[32] JaCoCo. Jacoco - java code coverage library. https://www.jacoco.org/jacoco/trunk/index.html, 2024. Accessed at January 1, 2024.

[33] Tao Jie. Datanodes usage is imbalanced if number of nodes per rack is not equal. https://issues.apache.org/jira/browse/HDFS-13279, 2017. Accessed at March 27, 2024.

[34] Jkroonza. Massive latency spikes (resulting in trashing and requiring a force-remount to resolve). https://github.com/gluster/glusterfs/issues/3356, 2020. Accessed at March 27, 2024.

[35] Abhay Kaviya. Keeping your Cisco DNA center healthy). https://www.cisco.com/c/en/us/td/docs/switches/datacenter/Cloud-Services-Platform/csp_5000/sae/release_notes/sae-sol-release-notes-2-2.pdf, 2024. Accessed at March 27, 2024.

[36] kemptechnologies. DNS, load balancing and DDOS attacks. https://kemptechnologies.com/blog/load-balancing-and-ddos-attacks, 2024. Accessed at January 1, 2024.

[37] The Linux Kernel. FUSE. https://www.kernel.org/doc/html/next/filesystems/fuse.html, 2024. Accessed at January 1, 2024.

[38] 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*, pages 147–161, 2019.

[39] 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*, pages 147–161, 2019.

[40] Abhishek Lekshmanan. CephFS at CERN in view of disaster recovery). https://fosdem.org/2024/schedule/event/fosdem-2024-3298-cephfs-at-cern-in-view-of-disaster-recovery/, 2024. Accessed at March 27, 2024.

[41] LeoProject. LeoFS - a storage system for a data lake and the web. https://github.com/leo-project/leofs, 2024. Accessed at January 1, 2024.

[42] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[43] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. Perseus: A fail-slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 49–64, 2023.

[44] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[45] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Dirk Merkel et al. Docker: lightweight Linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[47] Atin Mukherjee. DHT-rebalance: Rebalance hangs on distribute volume when glusterd is stopped on peer node). https://bugzilla.redhat.com/show_bug.cgi?id=1245142, 2020. Accessed at March 27, 2024.

[48] NewDund. Delete a storage node would cause data loss. https://github.com/leo-project/leofs/issues/1115, 2018. Accessed at March 27, 2024.

[49] NetApp ONTAP. ONTAP: Data management software for a better hybrid cloud experience. https://www.netapp.com/data-management/ontap-data-management-software/, 2024. Accessed at January 1, 2024.

[50] Red Hat Customer Portal. Red hat gluster storage. https://access.redhat.com/products/red-hat-storage, 2024. Accessed at March 27, 2024.

[51] Christopher Salls, Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Exploring abstraction functions in fuzzing. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020.

[52] IBM Storage Scale. Accelerate AI with a global data platform and break through data barriers. https://www.ibm.com/products/storage-scale, 2024. Accessed at January 1, 2024.

[53] 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, Vancouver, BC, August 2017. USENIX Association.

[54] Deepak C Shadrach, Kiran S Balagani, and Vir V Phoha. A weighted metric based adaptive algorithm for web server load balancing. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 1, pages 449–452. IEEE, 2009.

[55] 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.

[56] Ravideep Singh, Pradeep Kumar Gupta, Punit Gupta, Reza Malekian, Bodhaswar T Maharaj, Darius Andriukaitis, Algimantas Valinevicius, Dijana Capeska Bogatinoska, and Aleksandar Karadimce. Load balancing of distributed servers in distributed file systems. In *ICT Innovations 2015: Emerging Technologies for Better Living 7*, pages 29–37. Springer, 2016.

[57] Leonid Usov. Inconsistent usage of the return codes in the mds code base. https://tracker.ceph.com/issues/64611, 2021. Accessed at March 27, 2024.

[58] Kenneth Waegeman. Ceph mdss keep on crashing within the rebalance process. https://tracker.ceph.com/issues/41935, 2021. Accessed at March 27, 2024.

[59] 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 symposium on Operating systems design and implementation*, pages 307–320, 2006.

[60] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 122–es, 2006.

[61] Udi Wieder et al. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[62] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[63] Xuehan Xu. IO hangs when issuing balanced/localized reads to replica crimson osds while the pg is still peering. https://tracker.ceph.com/issues/65806, 2021. Accessed at March 27, 2024.

[64] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing SMT solvers via two-dimensional input space exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–335, 2021.

[65] Tsozen Yeh and Chiahung Sun. Enhancing the reliability of cloud data through identifying data inconsistency between cloud systems. *Information Systems Frontiers*, pages 1–9, 2023.

[66] Yeshan333. A library for run-time system code line-level coverage analysis. you can use it to evaluate the intergration test coverage. https://github.com/yeshan333/ex_integration_coveralls, 2024. Accessed at January 1, 2024.

[67] Jianwei zhang. Imbalance load caused by effects of hdd/ssd on op latency and bandwidth when use mclock_scheduler. https://tracker.ceph.com/issues/63014, 2021. Accessed at March 27, 2024.

[68] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.