Finding Metadata Inconsistencies in Distributed File Systems via Cross-Node Operation Modeling

Fuchen Ma
Tsinghua University

Yuanliang Chen
Tsinghua University

Yuanhang Zhou
Tsinghua University

Zhen Yan
Tsinghua University

Hao Sun
ETH Zurich

Yu Jiang[*]
Tsinghua University

Abstract

Metadata consistency is crucial for distributed file systems (DFSes) as it ensures that different clients have a consistent view of the data. However, DFSes are inherently error-prone, leading to metadata inconsistencies. Though rare, such inconsistencies can have severe consequences, including data loss, service failures, and permission violations. Unfortunately, there is limited understanding of metadata inconsistency characteristics, let alone an effective method for detecting them.

This paper presents a comprehensive study of metadata inconsistencies over the past five years across four widely-used DFSes. We identified two key findings: 1) Metadata inconsistencies are mainly triggered by interrelated cross-node file operations rather than system faults. 2) The root cause of inconsistencies mainly lies in the metadata conflict resolution process. Inspired by these findings, we proposed *Horcrux*, a highly effective fuzzing framework for detecting metadata inconsistencies in DFSes. *Horcrux* uses cross-node operation modeling to reduce the infinite input combinations to a manageable space. In this way, *Horcrux* captures implicit cross-node operation relationships and triggers more conflict resolution logic. Currently, *Horcrux* has detected 10 previously unknown metadata inconsistencies. In addition, *Horcrux* covers 20.29%-146.21% more conflict resolution code than state-of-the-art tools.

1 Introduction

Distributed File systems (DFSes) enable users to access file storage from multiple devices as if they were accessing local storage. In a DFS, metadata is responsible for recording the fundamental elements of the file system, such as file structures and data properties. Unlike DFSes like HDFS [4], Lustre [5], and OrangeFS [49], which manage metadata in a central node, CephFS [61] and GlusterFS [19] adopt a distributed approach to manage metadata. It is crucial to ensure that the metadata retrieved by different clients and nodes meets the requirements of eventual consistency [6].

[*]Yu Jiang is the corresponding author.

In a DFS, file operations initiated by different clients can be executed concurrently across various distributed nodes. These operations are referred to as cross-node file operations. Cross-node file operations spread across the DFS nodes, introducing additional challenges in maintaining metadata consistency. To keep metadata consistent, a DFS adopts four steps: 1) Local Metadata Update: each node updates the metadata locally according to client's request. 2) Metadata Propagation: each node broadcasts metadata updates to others. 3) Conflict Resolution: resolve the metadata updating conflicts. 4) Metadata Commit: each node store the consistent metadata. Bugs in each step may result in metadata inconsistencies.

In a DFS, metadata inconsistencies can lead to severe consequences, including data losses and service failures. For example, the Greek Research and Technology Network (GRNET) uses CephFS as the storage backend for its cloud service, OKEANOS [27]. On October 14, 2016, a metadata inconsistency occurred in CephFS, causing six OSDs to mistakenly identify innocent nodes as crashed [34]. This incident ultimately led to widespread failures in the client's I/O and a significant outage in OKEANOS's VM services. Thus, understanding the causes of metadata inconsistencies and how to effectively detect them is crucial for the stability and security of a DFS. Unfortunately, despite the severe consequences, this issue has been lacking detailed study and analysis, let alone an effective detection method.

In this paper, we present an in-depth study of 44 real-world metadata inconsistencies over the past five years across four widely used DFSes, including CephFS [61], GlusterFS [19], LeoFS [36] and IPFS [2]. We analyzed the triggering conditions and root causes of these issues and got 2 key findings: ① Most metadata inconsistencies are triggered by interrelated distinct, cross-node file operations. We found that all of the bugs are triggered by cross-node file operations, while only a small percentag (2.27%) involve system faults. Specifically, distinct operators with potential relations trigger 63.64% of the inconsistencies. As for the operands, related file paths (parent/child) trigger 54.55% of the inconsistencies. ② The majority(63.64%) of the metadata inconsistencies are caused by flaws in the conflict resolution step. For example, bug#49912 in CephFS [9] has implementation bugs in the conflict resolution process for inodes and filenames. Another bug in GlusterFS [24] failed to properly handle the conflicts among rm operations on parent and child file nodes.

The existing DFS fuzzer Monarch [43] performs cross-node testing by mutating syscalls and faults under the guidance of code coverage. However, without exploring the implicit relationships, cross-node syscalls are unlikely to trigger these inconsistencies. Additionally, the code coverage guided fuzzing explores various code components rather than concentrating on the metadata conflict resolution part. Consequently, Monarch is not effective in identifying metadata inconsistencies. Other tools [26, 32, 58] aim at traditional file systems testing. However, these tools lack a holistic infrastructure to coordinate inputs across the nodes.

Findings ① and ② suggest that an effective metadata inconsistency detection tool should prioritize generating interrelated cross-node file operations that can trigger metadata conflict resolution. However, there are two main challenges to achieve that goal. **Challenge 1: Infinite cross-node file operation combinations.** File operations contain file operators and operands. Though file operators are countable, file operand is a vast array of file paths. Thus, the number of cross-node file operation combinations are virtually limitless. To model how these operations interact, we need to condense the exploration space into a manageable scope. **Challenge 2: Metrics for relationship mining.** Conflict resolving code in DFSes are complex and tightly coupled with other components. It's non-trivial to precisely identify the conflict resolving code coverage at runtime. Therefore, an easily accessible metric that can characterize conflict resolution logic is needed to evaluate whether cross-node operations are well correlated.

To address the aforementioned challenges and effectively detect metadata inconsistencies in DFSes, we propose *Horcrux*. Its key principle is modeling cross-node file operations and mining the relationships among them, guided by the duration of temporary inconsistencies. *Horcrux* constructs an operation relation table to abstract the workloads in DFSes and reduce the infinite exploration space of file operations into a tractable one. Each item in the table represents the relationship weight between two operations. In each fuzzing round, *Horcrux* calculates the duration of temporary inconsistencies, which is measured from the start of instruction processing to the final confirmation time. If a set of cross-node operations results in a longer temporary inconsistency duration, *Horcrux* considers it beneficial. The weights in such sets are increased to strengthen the relationships among those operations.

We have implemented and evaluated *Horcrux* on four commonly used DFSes. Totally, *Horcrux* has detected 10 previously unknown metadata inconsistencies. In addition, we compared *Horcrux* with state-of-the-art tools, including Monarch, Hydra, Syzkaller, and smallfile. The results show that other tools were able to find only 3 bugs detected by *Horcrux*. Furthermore, we evaluated the amount of conflict resolution logic triggered by *Horcrux* and other tools. The results show that *Horcrux* covers 20.29%-146.21% more conflict resolution code lines than state-of-the-art tools, indicating a higher likelihood of triggering metadata inconsistencies.

Our paper makes the following contributions:

- We conducted a thorough study of 44 historical metadata inconsistencies in DFSes and analyzed their severity, root causes, and triggering conditions.
- We proposed a method for cross-node operation modeling to mine implicit relationships, and implemented it in *Horcrux* to detect metadata inconsistencies in DFSes.
- We evaluated *Horcrux* on 4 DFSes, and detected 10 previously-unknown metadata inconsistencies. We will open-source *Horcrux* [1] for practical use.

2 Overview

2.1 Distributed File Systems

A distributed file system (DFS) is a core component in networked computing, enabling the storage, retrieval, and management of files across multiple interconnected servers or nodes. Unlike traditional file systems, the workloads in DFSes are handled by multiple distributed nodes. These workloads are referred to as cross-node file operations. In a DFS, metadata serves as the foundational framework, recording critical information such as file structures, access permissions, modification timestamps, and more. Many DFeSs store metadata in a distributed manner to ensure scalability and robustness. Consensus protocols are used to maintain the consistency among distributed nodes. However, consensus protocols incur significant performance overhead. For DFSes, performance trade-offs need to be made. Thus, techniques like consistent hashing are used to determine the metadata storage locations, ensuring consistency. However, in systems like Ceph, Paxos is utilized, but only during the synchronization of the MDS.

We now illustrate how DFSes keep metadata consistent by using CephFS as an example. CephFS [61] supports multiple active metadata servers (MDSs) to manage metadata and handle metadata read and write requests from different clients. In CephFS, each MDS is responsible for a distinct partition of the file tree to ensure load balancing. As a result, each MDS forwards requests to the corresponding MDS to handle operations on different files. However, considering that the size of metadata can reach the PB level in a complex system, CephFS uses an object storage cluster rather than the MDS to store metadata. This means that MDSs in CephFS can be viewed as caches for the metadata. After each MDS finishes handling parallel requests from different clients, it persists the metadata updates to the object storage cluster. Despite

[1] Horcrux at: https://anonymous.4open.science/r/Horcrux-F00C

different MDSs managing distinct metadata partitions, each client should have an identical metadata view, such as file structures, regardless of which MDS it connects to.

2.2 Fuzz Testing

Fuzz testing is a technique that generates a wide range of unexpected and invalid inputs to uncover vulnerabilities and bugs in a program. Essentially, a fuzzing process consists of four steps: 1) information collection, 2) seed selection, 3) seed generation, and 4) seed execution. Well-known fuzzers such as AFL [25], LibFuzzer [41], and Peach [16] follow these steps for testing. During the information collection phase, a fuzzer gathers key data from the program under test by instrumenting the code. This information is then used to identify bugs and guide seed selection. In the seed selection phase, the fuzzer chooses high-quality seeds or templates based on the collected feedback. In the seed generation phase, the fuzzer applies various strategies to produce new test inputs based on the selected seeds or templates. Finally, in the seed execution phase, the fuzzer feeds the new seed into the program and waits for execution to complete.

Fuzzing a DFS also follows the four steps outlined above. For information collection, a DFS fuzzer needs to gather execution data from multiple nodes and integrate the distributed information into a unified metric. In seed selection, the fuzzer should maintain a seed pool and choose high-quality seeds based on the collected information. For seed generation, the fuzzer needs to generate cross-node file operations. Additionally, the generation strategy for the operands should be tailored to the file structures of the DFS, such as by selecting existing files or directories. In the seed execution phase, the fuzzer sends the operations to the DFS with DFS clients, typically mounted using FUSE [13] or the Linux kernel [14]. We adhered to this workflow in the design of *Horcrux*.

2.3 Threat Model

Throughout this paper, we use the following threat model. *(1) Attackers' capabilities.* Under our threat model, attackers can control DFS clients and perform file I/O operations remotely. They cannot control any DFS node. *(2) Bug Definition.* We define the metadata inconsistencies as the view of the metadata achieved by each client is different. *(3) Attacks conduction.* Attackers can issue well-crafted file I/O operations through specific clients, causing other clients to observe inconsistent metadata views. This leads to symptoms such as data corruption, and may further escalate into severe consequences like DoS, data losses, or permission violations.

The justification for this model is that DFSes allow multiple clients to mount shared storage. For example, RedHat's OpenStack [48] uses CephFS for this purpose. Metadata inconsistencies, however, pose potential risks, enabling attackers to exploit them for malicious file I/O that can impact others.

3 Understand DFS Metadata Inconsistencies

3.1 DFS metadata consistency maintenance

Figure 1 shows the four steps involved in a DFS to maintain metadata consistency: local metadata update, metadata propagation, conflict resolution, and metadata commit.

In the example shown in the figure, three clients perform three file operations concurrently: mv /foo /foo1 ⇔ delete /foo/bar ⇔ chmod /foo/bar. Here, foo' and foo1' are directories, while 'bar' is a file. To maintain consistency, a DFS follows these steps: **1) Local metadata update.** Each node handles the file operation issued by each client and updates the metadata locally. In this example, node 0 updates the metadata for /foo', while node 1 and node 2 update the metadata for /foo/bar'. **2) Metadata propagation.** In this step, each node broadcasts its metadata updates to other nodes and receives updates from the others simultaneously. **3) Conflict resolution.** If the metadata updates conflict, DFSes must resolve them. In this case, the file '/foo/bar' is deleted in node 1, but still exists in node 0 and node 2. A DFS should resolve such conflicts to achieve a consistent state. To accomplish this, GlusterFS uses a healing process to select one node as the resolution source and sync the metadata updates to other nodes, while CephFS uses metadata servers (MDS) to handle conflicts based on the order of received requests. If node 0 is chosen by GlusterFS as the source, or the request is handled by the order 'chmod /foo/bar ⇒ mv /foo /foo1 ⇒ delete /foo/bar' of CephFS MDS, the resolution results will be '/foo1/bar'. **4) Metadata commit.** Finally, the consistent metadata is persisted. In GlusterFS, each node stores the metadata locally, while in CephFS, the MDS sends the metadata to the corresponding object storage devices (OSDs).

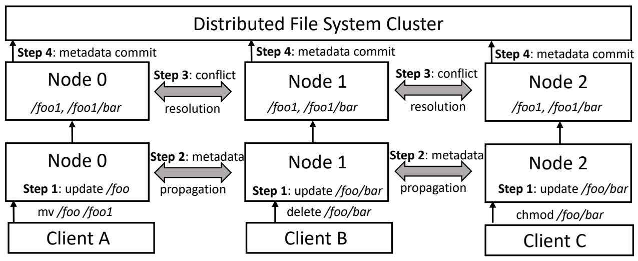


Figure 1: Steps for DFSes to maintain metadata consistency.

Temporary Inconsistency duration calculation. For DFSes, the nodes maintain metadata to achieve the eventual consistency. Thus, temporary inconsistency is allowed. In each DFS, the inconsistency duration can be calculated by the following way: $Dur_i = T_{consistent} - T_{issued}$. Here, $T_{consistent}$ means the first timestamp when every node in a DFS has the same view of the metadata. While T_{issued} represents the timestamp that the cross-node file operations are issued. The duration also equals to the summary of the duration of each phase in metadata consistency maintenance. For example, in Figure 1, T_{issued} is the timestamp when three clients issue the commands: mv /foo /foo1, delete /foo/bar, and chmod /foo/bar. And the timestamp $T_{consistent}$ is the time when step 4 is over, and each node shows the metadata view as '/foo1' and '/foo1/bar'.

3.2 Inconsistency Study Methodology

To better understand the characteristics of metadata inconsistencies, we choose 4 popular DFSes as our studying targets, including CephFS, GlusterFS, LeoFS, and IPFS. DFSes like HDFS, Lustre, and OrangeFS manage metadata in a central node, rather than in distributed manners. Thus, metadata inconsistency doesn't occur in them. So we did not choose them as the benchmark.

These targets are ubiquitous in DFSes just like ext3 [62] is ubiquitous in kernel file systems. CephFS is first proposed by Sage A. Weil, in OSDI 2006 [61]. According to the data from RedHat [7], Ceph storage is reliably the most popular storage for OpenStack with more than 50% market share. Based on the statistics [28], Ceph has been deployed in 6,119 well-known corporations, including SpaceX [57], and OpenStack [48]. GlusterFS is another DFS that can scale to several petabytes. According to statistics, 3,583 companies have used GlusterFS as an Enterprise Data Storage tool around the world [29]. LeoFS [36] is a distributed object/blob store system, which is able to store various kinds of raw data. While no public market share data is available, LeoFS has been adopted by several renowned enterprises, including Rakuten [53]. IPFS [30] is a decentralized DFS that uses content-addressing to uniquely identify each file in a global namespace. Lots of applications have been developed on the top of IPFS, such as a peer-to-peer database OrbitDB [50].

The comparison of these DFSes is shown in Table 1. CephFS is developed in C++. GlusterFS is implemented in C. LeoFS is created in Erlang, and IPFS is written in Go. For access protocols, CephFS, GlusterFS and LeoFS support POSIX file commands, while IPFS can only be accessed by the IPFS API. CephFS, GlusterFS, and LeoFS are widely used in cloud storage and high performance computing. IPFS is used in Web3 applications and blockchain systems. To store the metadata, CephFS use distributed Metadata servers. GlusterFS, LeoFS, and IPFS use DHT algorithms. For node types, CephFS support mulitiple types of nodes, including MDS (metadata servers), OSD (object storage devices), MON (monitors), and etc. While nodes in other three DFSes have the same type. An in-depth study of metadata inconsistencies on these targets represent a common feature of such problems.

Inclusion and exclusion for bug selection. We select bugs from the issue trackers of the above mentioned systems. 1) Inclusion criteria: we search for resolved and valid bugs whose titles contain the words 'metadata', 'view', 'mismatch', and

	CephFS	GlusterFS	LeoFS	IPFS
Language	C++	C	Erlang	Go
Access Protocol	POSIX	POSIX	POSIX	IPFS API
Use Cases	Cloud storage, HPC	Big data, cloud storage	cloud storage	Web3, blockchain
Metadata Storage	MDS	DHT	DHT	DHT
Node type	MDS, OSD MON,etc	Full node	Full node	Full node

Table 1: Detailed comparison of the targeted DFSes. DHT stands for dynamic hash tables. MDS is the metadata server, OSD is the object storage device, and MON is the monitor.

'inconsistency/inconsistent'. The issues should be flagged as 'bug' or 'vulnerability'. 2) Exclusion criteria: we manually checked the reports to exclude issues unrelated to metadata inconsistencies. We also excluded the bugs with: i) duplicate reports of the same issue. ii) reports that were misclassified as bugs but were actually feature requests or enhancements. Finally, we get 44 bugs listed in Table 2.

CephFS	GlusterFS	LeoFS	IPFS	Total
10	17	13	4	44

Table 2: Analyzed metadata inconsistencies numbers.

It should be mentioned that our analysis focuses on bugs from the past five years rather than the entire bug history. This selection ensures relevance to the current systems and practices. While the resulting set of bugs appears relatively niche in terms of quantity, these metadata-related issues are of significant severity, often leading to serious system failures.

3.3 Inconsistency Study Findings

3.3.1 Severity

We found that metadata inconsistencies are significantly more critical than other bugs in DFSes in three key aspects:

Urgency. 65.91% of the metadata inconsistency issues are classified as 'High-Priority' compared to only 6.47% for other types of bugs. This includes issues marked as 'Critical' or flagged with other urgency labels. The reason for this discrepancy may be that metadata inconsistencies are directly related to user-facing functionality. If left unaddressed, they can be easily exploited maliciously.

Consequences. We found that the majority of (54.55%) the metadata inconsistencies result in data loss (e.g., LeoFS bug#702 [60]), 27.27% lead to service unavailability (e.g., GlusterFS bug#919 [22]), and the remainder cause permission violations (e.g., CephFS bug#63906 [42]). This indicates that, although the relative number of metadata inconsistencies is small, their consequences are severe. For example, bug#1115 [46] in LeoFS causes different clients to see inconsistent views of the data. In some client views, data that should have existed appears to be deleted. As another example, bug#63906 [42] in CephFS can cause two clients to fetch inconsistent modes for the same file. One client may see the file mode as 32768 (indicating a regular file), while the other sees it as 32776 (indicating a file with root permissions). As a result, the second client may mistakenly gain elevated permissions on the file, potentially exposing credential data and leading to privacy leaks.

Impacts. We found that only 31.82% of metadata inconsistencies were detected before the corresponding versions were released, meaning the majority (68.18%) were exposed in production. This highlights the significant risks posed by such issues in live environments. Given that these bugs can be easily exploited and their potential consequences are severe, it is crucial to address them before releasing new versions.

3.3.2 Triggering Conditions.

Although most distributed system fuzzers [18, 44] inject system faults to detect bugs, we found that only one (2.27%) of the metadata inconsistencies are related to system faults. The case is the bug#112 in IPFS [66]. In that case, when two removed nodes re-enter the cluster, the state of certain files cannot be synchronized to them due to consensus failures. The reason may be that injecting faults is effective in detecting error-handling code bugs. According to our study, metadata inconsistencies do not occur in the error-handling processes. Thus, system faults are less likely to trigger metadata inconsistencies.

In contrast, all the metadata inconsistencies we analyzed require a set of cross-node file operations to expose. Furthermore, our study revealed two unique characteristics of such operations, shedding light on effective detecting tools:

1) Distinct operators. We found that distinct operators with potential relations, such as hardlink/create and create/mv, trigger 63.64% of the inconsistencies. Although the same operators trigger many inconsistencies as well, distinct operators that are related to each other are more effective in triggering metadata inconsistencies. For a testing tool, it's easy to construct cross-node operations with the same operators. However, tapping into the underlying relationships between distinct operators will likely make a testing tool more effective and stand out.

2) Related operands. As for the operands, operations involving related files (parent/child file nodes) trigger 54.55% of the metadata inconsistencies. While operations with exactly the same operands also trigger some bugs, generating cross-node operations with related operands can significantly improve the effectiveness of metadata inconsistencies detection.

3.3.3 Root Causes.

We analyzed the root causes of metadata inconsistencies based on the four steps required to maintain consistency. Our analysis revealed that the majority of metadata inconsistencies (63.64%) are caused by conflicts during the conflict resolution phase. Additionally, 22.73% are attributed to issues in the local metadata update phase, and 9.09% result from bugs in metadata propagation. The remaining two cases (4.55%) are caused by flaws in the metadata commit phase. To better understand how these flaws lead to metadata inconsistencies, we will present examples for each root cause.

Conflict Resolution Bugs. The majority of metadata inconsistencies stem from this issue. For example, bug#49912 in CephFS [9] causes different files to point to the same inode during the resolution of renaming conflicts. (An inode is a data structure in a Unix-style file system that describes a file-system object such as a file or directory.) This bug can lead certain threads to mistakenly assume they can safely acquire the lock and update the data, which may result in data loss. Another case is the bug#3677 in GlusterFS [24]. In that case, the DFS failed to handle the conflicts caused by cross-node file operations rm /B/D1 $\Leftrightarrow$ rm /B/D2 $\Leftrightarrow$ rm /B}.

Local Metadata Update Bugs. Another part of the root causes is the bugs in the local metadata update phase. For example, the bug#63906 of CephFS [42] occurs during the processing of '*chmod*' request locally and leads to metadata inconsistencies. The handling function returns before the '*chmod*' is actually accomplished.

Metadata Propagation Bugs. 4 cases are due to bugs in the metadata propagation phase. For example, the bug#51068 in CephFS [3] missed to sync some STS (subtree snap in CephFS) metadata properties such as roles and policies during metadata synchronization. As a result, various clients may not get inconsistent STS properties.

Metadata Commit Bugs. The last category of root causes for metadata inconsistencies is bugs in the metadata commit phase. For example, bug#1017 [47] in LeoFS results from implementation flaws in the data compaction process during metadata commitment. Similarly, bug#2845 [21] in GlusterFS is related to data flushing issues.

3.4 Requirements

Based on the findings, two key requirements can be derived for effectively detecting DFS metadata inconsistencies:

1. **Try to characterize and mine the relationships between cross-node file operations.** Based on the findings from the triggering conditions, the underlying relationships between cross-node operations play a crucial role in detecting metadata inconsistencies. An effective tool should model these relationships and learn them during the testing.

2. **Try to trigger more conflict resolution code.** According to the findings in root cause analysis, metadata conflict

resolving is the main reason for inconsistencies in DFSes. An effective detection tool should trigger more conflict resolution logic. A metric is required to evaluate which cross-node file operations are most effective at triggering such logic, guiding the generation of further test inputs.

4 Motivating Example

4.1 A Metadata Inconsistency in GlusterFS

We use an example of a metadata inconsistency in GlusterFS to illustrate how it happens and how to effectively detect it. The example is the bug#1546 in GlusterFS [23]. Figure 2 shows the workflows to trigger this bug. **Overall, this bug is caused by interrelated distinct cross-node file operations { create DC0 ⇔ touch DC0/F0 } due to a conflict resolution flaw.** DC0 here stands for a directory, and F0 means a file. In this case, three datacenters are sending requests to the GlusterFS cluster. The first datacenter sends an operation ‘create DC0’ to node 0. The second datacenter sends the operation ‘touch DC0/F0’ to node 1. The last datacenter sends ‘lookup DC0’ to node 2. These cross-node operations trigger both data and metadata synchronization. First, each node synchronizes the data. After this step, each node contains the directory ‘DC0’. However, for nodes 1 and 2, none of the extended attributes (xattrs) are set on the directory ‘DC0’ until the metadata sync is completed. Here, xattrs represent extended file attributes, which are filesystem features that allow users to associate computer files with metadata that is not interpreted by the filesystem itself. At this point, the cluster handles the ‘touch’ request and sets a default xattr ‘root’ on the directory ‘DC0’ of node 1.

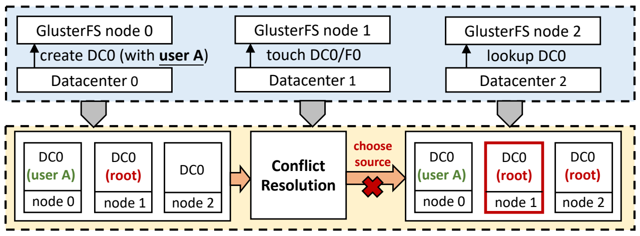


Figure 2: A metadata inconsistency in GlusterFS due to conflict resolution flaws, which is triggered by distinct cross-node operations { create DC0 ⇔ touch DC0/F0 }.

During the metadata synchronization process, conflicts among the xattrs across nodes should be resolved. To address these conflicts, GlusterFS selects one node as the synchronization source. However, a bug exists where the cluster mistakenly selects node 1 as the source, despite the fact that the xattr for the entry ‘DC0’ is just a default value: ‘root’. This incorrect value gets synchronized, resulting in a situation where only datacenter 0 sees the permission of ‘DC0’ as ‘user A’, while the other datacenters incorrectly see the permission as ‘root’. This bug has been fixed by the corresponding developers [31]. They implemented an ‘outcast’ logic in the metadata sync process, which prevents entries that have not completed the initial metadata synchronization from being selected as the source for conflict resolution.

4.2 Challenges and Our Key Insights

Limitations of existing tools. Existing tools have limitations in generating interrelated cross-node operations that trigger the deep logic of metadata conflict resolution in DFSes. In this case, the relationships with the file operations are essential to trigger this bug. Specifically, the operator ‘create’ set the permission to ‘user A’ while the operator ‘touch’ set the permission to ‘root’. Thus, the cross-node operations { create DC0 ⇔ touch DC0/F0} are key to this inconsistency.

However, existing DFS fuzzing tool Monarch [43] mutates syscalls and system faults without knowing the implicit relationship of these operations. Besides, the faults are not helpful to trigger the bug in this case. While traditional file system testing tools such as Hydra [32], Healer [58] and Syzkaller [26] failed to generate cross-node inputs, let alone to mine the relationships and trigger conflict resolution.

In this case, our design will constantly generate cross-node file operations like ‘create’, and ‘touch’. After the execution of each set of cross-node operations, our approach calculates the duration of metadata inconsistency. The cross-node operations {create DC0 ⇔ touch DC0/F0 ⇔ lookup DC0} will lead to longer inconsistency duration compared with operations like {create DC0 ⇔ create DC1 ⇔ lookup DC2}. Consequently, these operations are more likely to occur under our approach. Thus, we successfully generate such cross-node operations, ultimately detecting this bug.

5 Horcrux Design

Our key insights. To address the challenges faced by existing tools, we adopt a cross-node operation modeling approach and capture the relationships among operators and operands guided by the temporary inconsistency duration. Table 3 describes the comparison of our design and the existing works. Our approach generates cross-node fie operations to trigger deep logic of conflict resolution. The guidance in our design is the inconsistency duration. This is based on an insight that more complex conflict resolution processes lead to a longer period of temporary inconsistency. While other approaches use code coverage to guide the fuzzing process. Furthermore, only Monarch and our approach support generating cross-node inputs. And only our design generates conflict testing inputs in an effective way.

Why simply extending existing tools doesn’t work? To detect metadata inconsistencies in this case, a straightforward approach may be equip each DFS client with a single-node filesystem testing tool like Hydra and Syzkaller to generate

	Test Inputs	Guidance Metrics	cross-node inputs	conflict inputs
Monarch	system calls and faults	code coverage	✓	✗
Syzkaller	system calls	code coverage	✗	✗
Hydra	system calls and images	code coverage	✗	✗
smallfile	file operations	no guidance	✗	✗
Our Design	**file operations**	**inconsistency duration**	✓	✓

Table 3: Comparison between our design and other tools.

cross-node file operations. However, such cross-node operations are random, without relationships. In addition, each Hydra (or Syzkaller) instance are independent. Without a unified operation generator, interrelated cross-node operations cannot be conducted even we know the relationships.

For Monarch, a simple extension may be eliminating the system faults and adding inconsistency monitors. However, to learn the implicit relationships, a metric is needed to judge what cross-node file operations have good relations. And a model of cross-node operations is also needed to strengthen the relations for further testing inputs generation. Monarch uses the code coverage as the metric. It may direct the fuzzer to other code components rather than conflict resolution. And without a proper model, Monarch has no way to fetch the relationships among cross-node file operations. Besides, directly manipulate the inputs to the conflict resolving code through fault-injection is ineffective too. This approach may import false positives, because some input cases may not valid due to the upper-level function may checks the parameters.

Overall design. Figure 3 shows the overall workflow of *Horcrux*. 1) The combinations of cross-node file operations are virtually limitless. The combination can be calculated by the product of the operators' number and the operands' number, raised to the power of the clients' number. To reduce the infinite exploration space, *Horcrux* constructs a relation table by making abstractions of both file operators and operands. 2) To generate interrelated cross-node operations and trigger complex metadata conflict resolution, *Horcrux* leverages the inconsistency duration to update the relation table and generate high-quality cross-node workloads. The heuristic is based on the insight that more complex conflict resolution processes will lead to longer periods of temporary inconsistency. We will introduce the details of each step.

5.1 Cross-Node Operations Modeling

5.1.1 Operations Abstraction

In order to reduce the infinite combinations of cross-node file operations into a tractable one, *Horcrux* models the operations. In *Horcrux*, a file operation is defined as $\langle Opt_i, Opd_i \rangle$. Opt_i

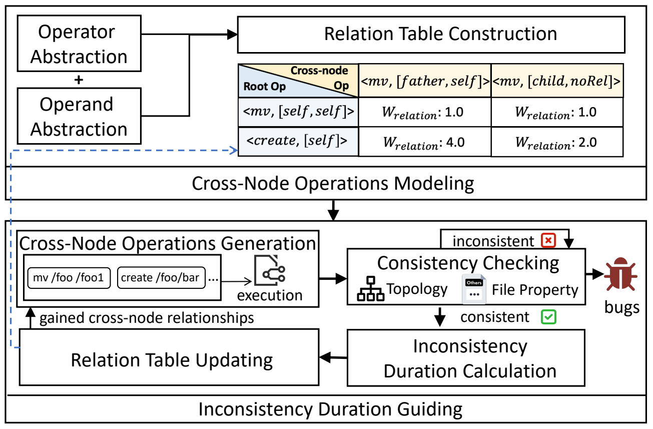


Figure 3: The overall workflow of *Horcrux*. *Horcrux* first constructs a cross-node file operation relation table for operation modeling. Afterward, *Horcrux* leverages the temporary inconsistency duration to guide the generation of the cross-node operations and try to trigger metadata conflict resolution.

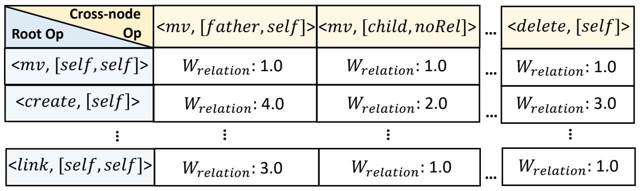
represents the operator of a file command such as 'mv'. Opd_i records the operands of the operator. Some operators have two operands, while others have only one. To accommodate this, we define Opd_i as a list $[opd_1, opd_2]$, where the second element may be None if the operator only has one operand.

Operator Abstraction. Considering that different DFSes tend to have distinct operators, constructing a unified model for all operators is challenging. However, we found that all operators can generally be divided into two categories: POSIX-standard operators and non-POSIX ones. For DFSes that support POSIX operators, such as CephFS, *Horcrux* only needs to model the POSIX operators. For other DFSes, like IPFS, we collect the operators based on their documentation.

Operand Abstraction. The operands of file operations are primarily file paths. Although file paths are virtually infinite, we found that the relationships among these paths can be abstracted into four types. The first type is $self$, which means the two operands are exactly the same. The second type is $parent$, which represents that one path is the parent of another. The third type is $child$, indicating that one path is a child of another. The last type is $noRel$, which means two paths have no relationships. With this abstraction, *Horcrux* reduces the infinite exploration space into a manageable one.

5.1.2 Operation Relation Table Construction

To characterize the relationships between cross-node file operations, *Horcrux* maintains a file operation relation table, as shown in Figure 4. As depicted in the figure, the rows of the relation table correspond to the root operations, while the columns represent the cross-node operations. A root operation is the initial operation chosen by *Horcrux* in each fuzzing round. Based on the root operation, *Horcrux* locates the corresponding row in the table and fetches (n-1) cross-node operations from that row, where ‘n’ represents the number of clients during testing. The $W_{relation}$ in the table indicates the relation weight between operations. Initially, all the relation weights are set to 1. A higher weight in a given row increases the likelihood of choosing that operation during the fuzzing.



Root Op \ Cross-node Op	<mv, [$father, self$]>	<mv, [$child, noRel$]>	...	<$delete$, [$self$]>
<mv, [$self, self$]>	$W_{relation}$: 1.0	$W_{relation}$: 1.0	...	$W_{relation}$: 1.0
<$create$, [$self$]>	$W_{relation}$: 4.0	$W_{relation}$: 2.0	...	$W_{relation}$: 3.0
⋮	⋮	⋮		⋮
<$link$, [$self, self$]>	$W_{relation}$: 3.0	$W_{relation}$: 1.0	...	$W_{relation}$: 1.0

Figure 4: The file operation relation table of *Horcrux*. $W_{relation}$ in the table represents the relation weights between cross-node operations.

When querying the table, *Horcrux* will replace the concrete operands with the abstraction $self$. For example, if the current root operation is $\langle mv, [A, B] \rangle$, *Horcrux* will match it to the row $\langle mv, [self, self] \rangle$. For the operand opd_1, there are three possible choices: $self$, $parent$, $child$. $self$ selects the same file in the root operation, $parent$ chooses the parent node of the file, and $child$ selects one of the children. While the operand opd_2 can be chosen from $\{self, parent, child, noRel\}$, where $noRel$ randomly chooses an existing file or directory or randomly generates a new file.

5.2 Inconsistency Duration Guiding

5.2.1 Cross-Node File Operation Generation

Horcrux generates cross-node operations based on the file operation relation table. First, it randomly produces a root operation. For all supported operators, *Horcrux* randomly select one. According to the operator, *Horcrux* generates the corresponding operands by randomly selecting an existing file/directory or generating a new file/directory.

After generating the root operation, *Horcrux* generates the other cross-node operations by referencing the relation table. The number in the table represents the probability of the corresponding operation being chosen. In this way, *Horcrux* selects several operations from the table as the cross-node operation templates. Based on these templates, *Horcrux* generates the concrete operators and operands. For operator generation, *Horcrux* chooses the operator provided by the template. For example, if the table gives a template $\langle mv, [self, noRel] \rangle$, the chosen operator will be mv. For operands generation, *Horcrux* uses the length of operands in the root operation and the template to categorize the strategies.

The same length. If both the root and template operations have one operand, *Horcrux* will generate operands based on the template's guidance. If the template operand is $self$, the generated operation will have the same operand as the root operation. If the template operand is $parent$, *Horcrux* will select the parent node of the operand in the root operation. For a $child$ operand in the template, *Horcrux* will randomly choose one of the children files. If both the root and template operations have two operands, the first operand is generated in the same manner. The second operand may also include $noRel$. The $noRel$ operand instructs *Horcrux* to randomly choose an existing file or directory, or to generate a new one.

Lengths are 1 and 2. If the root operation has one operand and the template operation has two operands, *Horcrux* will adapt the template to match the root operation's structure by keeping the same operator and first operand. The second operand in the template will be set to $noRel$. Thus, for the first operand, *Horcrux* will choose the operand in the root operation itself, its parent, or its child. For the second operand, *Horcrux* will either select an existing file/directory or randomly generate a new one.

Lengths are 2 and 1. If the root operation has two operands and the template operation has one, *Horcrux* will generate the operand based on the first operand of the root operation.

After generating the cross-node operations, *Horcrux* sends them to the DFS cluster via various clients. Then *Horcrux* waits for the execution to finish, and checks whether a metadata inconsistency exists. If an inconsistency is found, *Horcrux* reports the corresponding issues and exits the fuzzing process. Otherwise, it collects the current metadata consistency maintenance logs and calculates the duration to further guide the updating of the relation table.

5.2.2 Consistency Checking

To check whether metadata has achieved consistency, we first model the metadata in a DFS. Metadata typically describes basic information about a DFS, such as file structures and data properties. Thus, we define metadata in DFSes as a rich tree that contains both topology information and property information. A node in the metadata tree is defined as $\langle C, P \rangle$, where C represents the child list of that node, and P denotes the property of the current file or directory. In BNF grammar [64], we define C and P as:

$$<C>::=<child> \mid <child>,<C> \mid \emptyset$$

$$<P>::=<name>,<authority>,<time>$$

Children list C can be an empty set, or a set of child nodes. Each node is identified by its inode. If a node has multiple children, they are sorted according to the alphabetical order of their names. The property P contains three elements: $name, authority, time$. $name$ represents the node's name, while $authority$ indicates the permissions held by each user on this node, such as who can write to this file and who is the owner of this file. The property $time$ records the last modification time, access time, and creation time.

According to the metadata definition, we define two monitors for checking metadata inconsistencies: the topology monitor and the property monitor. **1) Topology monitor:** This monitor checks whether different clients observe the same metadata tree topology. It does this by traversing the tree and ensuring that the child list C of each node is consistent across all clients. **2) Property monitor:** This monitor checks whether all three properties of a node are the same from multiple clients. For each node, it first verifies that the name is the same across clients. For example, a historical bug in CephFS [9] violated this assertion. Next, it checks whether the authority properties are consistent across clients. This includes verifying the owner of the file and the permissions for each user. A bug in CephFS [17] caused a metadata inconsistency regarding layout permissions on the root directory, violating this rule. Finally, the monitor checks the time properties of each node. Multiple clients compare modification, access, and creation times. A bug in CephFS [37] caused a directory's modification time to be overwritten during a update cap request when creating directories, breaking this monitor. In each fuzzing round, *Horcrux* verifies metadata consistency. After issuing cross-node file operations, it records a timestamp. If both monitors confirm that the metadata is consistent, *Horcrux* records a second timestamp. Using these timestamps, it calculates the duration of the inconsistency.

Bugs Identification. If all monitors confirm that the metadata is consistent, *Horcrux* will proceed with the next round of fuzzing. However, if any monitor detects metadata inconsistencies, *Horcrux* will not immediately classify it as a bug. Given the complex conflict resolution mechanisms, nodes in DFSes may take some time to achieve eventual consistency. Therefore, *Horcrux* will continue to recheck the consistency using the two monitors. If the metadata remains inconsistent, *Horcrux* will then classify it as a bug.

5.2.3 Operation Relation Table Updating

To handle the seeds that trigger more conflict resolving logic, *Horcrux* updates the relation table weights for a high possibility for further generation. Based on the calculation of the inconsistency duration, *Horcrux* determines whether the current cross-node file operations are good. In the fuzzing process, *Horcrux* will record the maximum inconsistency duration D_m. D_m is dynamically updated. If a longer inconsistency duration is found, we will update the D_m to restore the maximum duration. And *Horcrux* will consider the operations as good ones. For good cross-node operations, *Horcrux* recognizes the first operation as the root operation and updates the values in the relation table according to other operations.

Algorithm 1 outlines the process of updating the file operation relation table in *Horcrux*. The input of the algorithm contains the original relation table, the current cross-node file operations, and the metadata tree. The output is the updated relation table. First, as shown in line 1, *Horcrux* fetches the

Algorithm 1: File Operation Relation Table Updating

```
Input  : $T_{org}$: Original file operation relation table
         $O_c$: Cross-node file operations
         $Tree$: The metadata tree
Output : $T_{new}$: New file operation relation table
1  $T_{new}$=$T_{org}$, $root\_op$ = $O_c[0]$;
2  for each op in $O_c[1:]$ do
3      $opr$ = [], $opt$ = extractOpt($op$);
4      $opr\_root$ = extractOpr($root\_op$);
5      $opr\_op$ = extractOpr($op$);
6      if $len(opr\_root) == len(opr\_op)$ then
7          $opr$ = cmpOpr($Tree, opr\_root, opr\_op$);
8      end
9      if $len(opr\_root) == 1$ then
10         $opr\_1$ = cmpOpr($Tree, opr\_root, opr\_op[0]$);
11         $opr$ = [$opr\_1$, $noRel$];
12     end
13     if $len(opr\_root) == 2$ then
14         $opr$ = cmpOpr($Tree, opr\_root[0], opr\_op$);
15     end
16     $target\_op$ = join($opt, opr$);
17     $Tree[root\_op][target\_op]$ += 1;
18 end
```

root_op. For other operations, *Horcrux* extracts the operators and the operands as described in lines 3-5. The length of *opr_root* and *opr_op* can be 1 or 2 (operators like 'create' have one operand and 'mv' has two operands).

If *opr_root* and *opr_op* have the same length, as shown in lines 6-8, *Horcrux* directly determines the relation between the operands from *opr_root* and *opr_op*. This relation could be 'parent', 'self', 'child', or 'noRel'. As lines 9-12 indicate, if the length of *opr_root* is 1 and the length of *opr_op* is 2, *Horcrux* will only calculate the relation between the first operands, which is stored as opr_1. Unlike the *opr* in line 7, the second operand in the list *opr* will be set to *noRel*, since the second operand in opr_root is none As line 14 shows, if the length of *opr_root* is 2 and the length of *opr_op* is 1, *opr* will only be calculated by the first operands in *opr_root* and *opr_op*. *Horcrux* combines *opt* and *opr* to determine the target operation *target_op* in line 16. As line 17 shows, *Horcrux* updates the *target_op*'s item in the table.

We now use three examples to illustrate this process, categorized based on the lengths of *opr_root* and *opr_op*.

- **The same length.** Suppose *opr_root* is $\langle cp, [A, B]\rangle$ and *opr_op* is $\langle link, [C, B]\rangle$, where A is the parent of C. Since both operations have two operands, the *opr* at line 7 will be $[child, self]$. The *target_op* will be $\langle link, [child, self]\rangle$. *Horcrux* will increment the value at row $\langle cp, [self, self]\rangle$ and column $\langle link, [child, self]\rangle$ in the relation table by 1.

- **Lengths are 1 and 2.** Suppose *opr_root* is $\langle create, [A]\rangle$ and

opr_op is $\langle link, [C,B] \rangle$, where A is the parent of C. The opr at line 11 will be $[child, noRel]$ and the $target\_op$ will be $\langle link, [child, noRel] \rangle$. *Horcrux* will increment the value at row $\langle create, [self] \rangle$ and column $\langle link, [child, noRel] \rangle$ by 1.

- **Lengths are 2 and 1.** Suppose opr_root is $\langle cp, [A,B] \rangle$ and opr_op is $\langle delete, [A] \rangle$. The opr at line 14 will be $[self]$ and the $target\_op$ will be $\langle delete, [self] \rangle$. *Horcrux* updates the value at row $\langle cp, [self, self] \rangle$ and column $\langle delete, [self] \rangle$.

6 Implementation

Figure 5 illustrates the implementation details of *Horcrux*. To ensure adaptability to different platforms, the implementation is divided into three main parts. The first part is the *Horcrux* Fuzz Engine, which includes a cross-node operation generator and a file operation relation table. These components are consistent across all DFS platforms. The second part is the Mount component, responsible for mounting the DFS under test and forwarding the file operations generated by *Horcrux* to the DFS cluster. This part provides two mounting methods: the kernel FUSE module and command modeling. By using FUSE, *Horcrux* can call POSIX APIs [63] directly on DFSes like CephFS, GlusterFS, and LeoFS. For DFSes that do not support FUSE (such as IPFS), *Horcrux* models the file commands of the DFS by writing a JSON file that records each command's operator and operands.

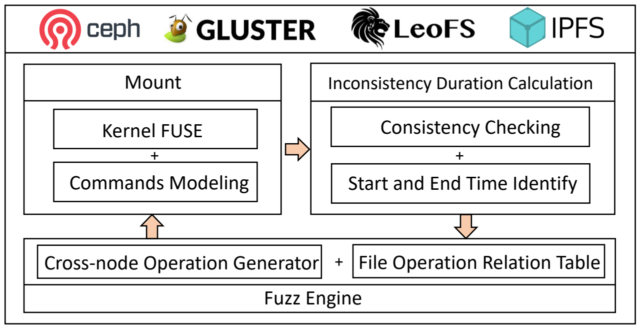

ceph
GLUSTER
LeoFS
IPFS
Mount
Kernel FUSE
+
Commands Modeling
Inconsistency Duration Calculation
Consistency Checking
+
Start and End Time Identify
Cross-node Operation Generator
+
File Operation Relation Table
Fuzz Engine

Figure 5: The implementation of *Horcrux*. is divided into three parts on 4 DFSes.

For metadata inconsistency detection, *Horcrux* uses different ways according to the mount methods. If the DFS is mounted by FUSE, metadata checking can be accomplished by using commands like 'tree' (for file structure) or 'll' (for properties). Otherwise, *Horcrux* uses platform-specific commands to check metadata. For example, for IPFS, *Horcrux* uses 'ipfs-cluster-ctl status' to check the property of each file.

7 Evaluation

To evaluate the effectiveness of *Horcrux*, we compared it with state-of-the-art file system testing tools, including Monarch [43], Hydra [32], Syzkaller [26] and smallfile [1]. The experiments are conducted on four widely used distributed file systems which contain CephFS [61], GlusterFS [19], LeoFS [36] and IPFS [2]. We did not choose HDFS, Luster or OrangeFS because they manage the metadata in a central manner. We ran each distributed system in a cluster on 10 virtual machines. The servers are deployed in our lab, allowing us to customize the experimental environment. Each machine has a 2.25 GHz 128-core CPU(with sufficient resources, the number of cores has little impact on the results of this experiment), 128 GB of RAM, and a 500 GB SATA SSD. They are connected with each other via a 10 Gbps bandwidth network. All the machines are operated by Ubuntu 22.04.3 with Linux kernel version 5.15.0. For CephFS, we deployed 5 monitor nodes, 5 metadata servers on 5 virtual machines, and 10 object storage devices. For GlusterFS and IPFS, we deployed 10 nodes on the 10 machines. For LeoFS, we deployed 3 manager nodes and 7 data storage nodes. Besides, we built 5 clients for each DFS where each client connects directly to one node in the DFS. All the experiments are conducted 5 times, and the results use the average values. We designed the experiments to address the following research questions:

- **RQ1:** Is *Horcrux* effective in detecting metadata inconsistencies in DFSes?

- **RQ2:** Can *Horcrux* trigger more metadata conflict resolution logic compared to other state-of-the-art tools?

- **RQ3:** Is inconsistency duration guidance effective?

7.1 Metadata Inconsistencies Detection

Horcrux has detected 10 previously unknown metadata inconsistencies in the commonly-used DFSes. Specifically, we found 3 inconsistencies in CephFS, 4 in GlusterFS, 2 in LeoFS and 1 in IPFS. 7 metadata inconsistencies are of the property inconsistency type, and 3 are of the topology inconsistency type. Bug #2 occurs in the local metadata update phase. Bug #3 occurs in the metadata propagation phase. And other 8 bugs are found in the conflict resolution phase.

Table 4 shows the details of the metadata inconsistencies found by *Horcrux*. We also used the workloads generated from Monarch, Hydra, and Syzkaller to test these DFSes. For smallfile, it can generate distributed file operations for various hosts. For Hydra and Syzkaller, we equipped each client with a workload generator provided by them. As a result, Monarch, Hydra and Syzkaller found 3 of the metadata inconsistencies detected by *Horcrux* (Bug #5, Bug #6, and Bug #9). Monarch finds 3 bugs mainly because the file operations it generates are not inter-related, and thus cannot effectively triggering conflict resolution code, which is the main reason for metadata inconsistencies. While smallfile found 0 bugs.

Table 4: Previously-unknown metadata inconsistencies found by *Horcrux* in 24 hours on 4 widely-used DFSes. *Horcrux* detected 3, 4, 2, and 1 bugs in CephFS, GlusterFS, LeoFS, and IPFS, respectively. The identifiers are anonymized for double-blind review.

#	Platform	Bug Type	Bug Description	Identifier
1	CephFS	Property Inconsistency	Writing to a file in parallel, some clients report 0 sizes, while others report over 200 MB.	Bug#63013
2	CephFS	Property Inconsistency	Cephfs different nodes have a different view of the REPORTED state due to interface bugs.	Bug#62704
3	CephFS	Property Inconsistency	A metadata server map decoding bug during PAXOS sync processes, leads to service failures.	Bug#65423
4	GlusterFS	Topology Inconsistency	The view of gluster nodes becomes inconsistent after a series of mutated file operations due to sync error.	Bug#4163
5	GlusterFS	Property Inconsistency	One of the nodes in the cluster becomes a read-only filesystem while other nodes remain normal.	Bug#4164
6	GlusterFS	Topology Inconsistency	After removing plenty of files, only the data of some nodes is deleted. The other nodes remains unchanged	Bug#4170
7	GlusterFS	Topology Inconsistency	Parallel executing chmod, rm, metadata became inconsistent due to the write-conflict error.	Bug#4182
8	LeoFS	Property Inconsistency	Parallel rewriting data with different data sizes but the checksum is not changed while data have changed.	Bug#1209
9	LeoFS	Property Inconsistency	INCONSISTENT HASH between the manager node and other nodes due to errors in synchronize_ring.	Bug#1211
10	IPFS	Property Inconsistency	Simultaneous execution of gc and add causes the gc to get stuck, leading to inconsistent file storage.	Bug#1993

Bug Severity. Metadata inconsistencies tend to cause severe consequences to the DFSes. Specifically, the bugs detected by *Horcrux* can lead to client I/O failure, data losses, and widespread service outages. 3 bugs can lead to the improper processes of client I/O requests, including Bug #1, Bug #5, and Bug #10. Clients cannot read or write any data to the file system which may affect the business logic of their applications and finally lead to property losses. For example, developers from Ceph triaged and confirm the Bug #1 that the vulnerability is related to file(lock) in the MDS and can lead to io hang. 3 bugs, including Bug #4, Bug #6, and Bug #7 may lead to data losses by losing some topology information of the file systems. A client may upload some files into the DFSes and cannot find them due to the inconsistency in metadata topology. 4 bugs, including Bug #2, Bug #3, Bug #8, and Bug #9 may lead to widespread service outage. These bugs corrupt some key properties like file hash or node statuses. This may influence the further handling logic of DFS nodes and finally lead to widespread service outages just like the incident in October 2016 [34] as we mentioned in Section I.

Bug Exploitation. The metadata inconsistencies can be exploited to conduct attacks. Attackers can control DFS clients and perform file I/O operations remotely to leverage these vulnerabilities. The prerequisite for exploiting these vulnerabilities is that the attacker needs to mount a file system to the DFS storage cluster. DFS is designed to support multiple users, so there are many real-world deployment scenarios where different users mount to the same storage cluster. For example, in the OKEANOS [27] case mentioned in Section 1, the client can access the backend storage of the GRNET. Similarly, in the OpenStack [48] service provided by Redhat, users can also mount to the same storage backend. Specifically, Bug#2, Bug#3, Bug#8, and Bug#9 can be exploited to conduct DoS attacks. Bug#6 and Bug#7 can be leveraged to cause data losses. Bug#1, Bug#5, and Bug#10 result in permission violations.

Bug Discover Time. Figure 6 describes the time for *Horcrux*, Monarch, Hydra, and Syzkaller to discover the bugs. We listed all the bugs in four DFSes in the same plot.

Horcrux detected six bugs (over half of all the bugs) within the first six hours, demonstrating its ability to effectively identify metadata inconsistencies in shallow consistency maintenance logic. In contrast, other bugs were buried deeper within the DFSes' code and were harder to trigger in the early stages. These issues took longer for *Horcrux* to uncover. For instance, it took 19 hours and 34 minutes to trigger Bug #1. In contrast, other tools detected three bugs hidden in the shallow logic but took more time than *Horcrux*. Specifically, Monarch, Hydra, and Syzkaller identified Bug #5, Bug #6, and Bug #9, respectively, with an average detection time of 3.8, 2.8, and 1.2 hours. While *Horcrux* used 3.3, 2.7, and 0.8 hours, respectively.

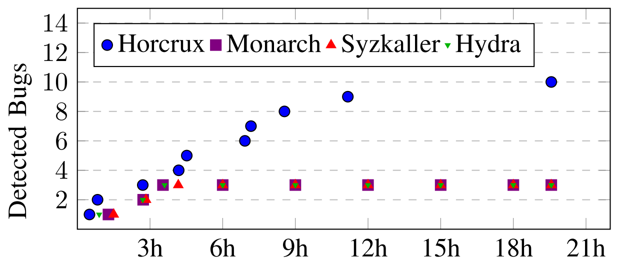


Figure 6: Bugs found by *Horcrux* and other tools over time. After 6 hours, the points of Monarch, Syzkaller, and Hydra coincide, as they find no new bugs beyond that time.

Bug Case Analysis. We now present a case to illustrate how *Horcrux* detects metadata inconsistencies in DFSes. The case involves Bug #4 listed in Table 4, a topology inconsistency bug found in GlusterFS. This bug prevents the resolution of metadata conflicts between nodes and leads to an unrecoverable state in the DFS. The code snippet in Figure 7 describes the detailed information of this metadata inconsistency.

Function 'glusterd_brick_sync()' is responsible for synchronizing the metadata of brick data among nodes when an inconsistency conflict occurs. Each time the brick synchronization process begins, an immediate service connection attempt is made, as shown in lines 9-13. However, this connection is a long-lasting channel that consumes significant resources. Once the brick synchronization is complete, the connection is closed, as shown in line 16. When the code is executed during the occasional metadata conflicts, everything

```
1  // sync to resolve the conflict among connected nodes
2  static int glusterd_brick_sync(...) {
3  +    if (!gf_is_service_connected(pidfile, &pid)) {
4  +        reconnect = true;
5  +    }
6  ...
7  connect:
8  +   if (reconnect) {
9        ret = gf_service_connect(brickinfo, socketpath);
10       if (ret) {
11         gf_msg(GF_ERROR, "Failed to connect to brick");
12         goto out;
13       }
14 +   }
15 out:
16     gf_service_close();
17     return ret;
18 }
```

Figure 7: A metadata inconsistency in GlusterFS. It stops conflict resolving between nodes and causes inconsistency.

works fine, and no bugs occur. However, intensive metadata conflicts occur when this function is called across several distributed storage nodes simultaneously where multiple clients operate the same data. And this triggers the brick data synchronization logic. Then the function ‘gf_service_connect’ is called at a high frequency. This results in the occupation of a large number of network resources, and blocks the normal metadata communication update of some nodes, and eventually leads to the inconsistency. Consequently, the GlusterFS service gets stuck, and key data cannot be synchronized, resulting in data losses. This bug has been fixed by adding a connection status checker, as shown in lines 3-5 and line 8.

In our experiments, this metadata inconsistency was detected solely by *Horcrux*. Normally, this bug cannot be triggered when the code snippet is executed on a single node. However, in the fuzzing environment created by *Horcrux*, a series of cross-node file operations with conflicts are sent to the storage nodes. As a result, nodes are likely to trigger the brick synchronization process repeatedly within a short time frame, ultimately leading to the bug. Exploiting this issue, attackers could initiate chaos attacks to bring down any normal node in the system.

Bug exploitation. Based on this bug, attackers can break down any normal node by conducting DoS attacks. This bug can be exploited by the following steps: 1) Attackers mount multiple clients to glusterFS nodes. 2) Attackers issue cross-node file operations from multiple clients simultaneously, access the same data and trigger the function ’glusterd_brick_sync’. 3) This bug is triggered and the whole cluster is stuck, leading to DoS attacks.

Historical Bug Evaluation We also adapt *Horcrux* and other tools on the previous versions of the 4 DFSes and evaluate whether they could successfully reproduce the 44 metadata inconsistencies we analyzed. Previous versions mean the versions where those bugs were not fixed. The result is shown in Table 5, *Horcrux* reproduced 31 bugs, Monarch, Syzkaller and Hydra reproduced 15, 10, and 10 bugs respectively, while smallfile reproduced 13 bugs. All bugs reproduced by other tools can be found by *Horcrux*.

Table 5: Historical bugs found by *Horcrux* and other tools.

Tool	CephFS	GlusterFS	LeoFS	IPFS	Total
Horcrux	6/10	12/17	10/13	3/4	31/44
Monarch	3/10	7/17	5/13	0/4	15/44
Syzkaller	2/10	5/17	3/13	0/4	10/44
Hydra	2/10	5/17	3/13	0/4	10/44
smallfile	3/10	6/17	4/13	0/4	13/44

The 13 inconsistencies that cannot be found by *Horcrux* are mainly due to two reasons. The first is that they are related to data races. These bugs can be hard to trigger because the race conditions are usually hard to be well-created in distributed systems. The other reason is that some bugs are related to specific requests beyond file operations. For example, the bug#1123 in LeoFS [51] requires the requests such as ‘reset-brick’ to trigger. *Horcrux* doesn’t support them for now, but they can be integrated into *Horcrux*’s model. To achieve this, developers can define a json file to describe the information of this command. The file contains the name, the number of arguments, the arguments’ types, the weight and the previous commands. We will introduce more details in Section 14.

```
1  relookup:
2    dn = dir->dir->dentries[dname];
3    ...
4    // Add the following code to fix the MIB
5  + if (!is_rename && dn->is_renaming) {
6  +   wait_on_list(waiting_for_rename);
7  +   goto relookup;
8  + }
9    ...
10   r = do_lookup(dir,dname,mask,target,perms);
```

Figure 8: The bug has been fixed by developers by checking whether the wanted dn is being renamed. As shown by lines 5-8, it waits for the renaming to be finished and relookup.

Only *Horcrux* can reproduce this bug. This is because it is challenging to create such conflict conditions effectively without understanding the relationships among cross-node file operations. The developers have already fixed this bug by adding a check statement to verify if another thread is renaming a file before proceeding with the lookup. The related code is shown in Figure 8. As shown in lines 5-8, the code checks whether the wanted dn is being renamed by another thread. If so, it will wait for the renaming to finish and relookup the d_n.

7.2 Conflict Resolution Logic Triggering

In this section, we evaluate whether *Horcrux* triggers more metadata conflict resolution logic compared to state-of-the-art tools by analyzing the conflict resolution code coverage. To perform this evaluation, we first instrumented the DFSes and collected the code line coverage data. For CephFS and GlusterFS, we used gcov [15] to collect the coverage information. Since LeoFS is written in Erlang, we utilized ExIntegrationCoveralls [65] for coverage collection. For IPFS, we employed goc [52] to gather the coverage data.

We collected the total code coverage of each tool (denoted as Cov_t) after 24 hours and manually identified the conflict resolution code coverage at the function level (denoted as Cov_c). The main reason is that the coverage and the conflict resolution logic coverage basically converged (with the increment of less than 2%). Besides, according to the best practice of fuzzing proposed by existing works [33, 54], 24 hours is a reasonable fuzzer runtime. We collected all the function names and parameters in the different covered codes. We determined whether the function is related to conflict resolution [2] based on its name and parameters. The results may not be absolutely accurate, but since these systems are mature open-source projects, functions names and parameters contain rich semantic information, making our statistics persuasive.

Table 6: The code coverage of *Horcrux* and other tools. Cov_t and Cov_c represent total coverage and conflict resolution related code coverage, respectively. '/' means the tool does not support the DFS.

	CephFS		GlusterFS		LeoFS		IPFS	
	Cov_t	Cov_c	Cov_t	Cov_c	Cov_t	Cov_c	Cov_t	Cov_c
Horcrux	57,357	15,981	38,446	14,535	8,067	1,168	6,530	1,135
Monarch	42,401	/	37,311	/	/	/	/	/
Syzkaller	34,026	7,229	33,236	9,968	6,370	892	/	/
Hydra	33,504	7,081	31,051	9,315	6,116	844	/	/
smallfile	37,530	8,246	34,229	10,611	6,805	971	/	/

The conflict resolution code coverage is shown in Table 6. Monarch does not support LeoFS and IPFS, while Syzkaller, Hydra, and Smallfile do not support IPFS. Additionally, Monarch only released the code image without symbols, making it impossible to identify the specific code lines it covered or calculate Cov_c. As the data demonstrates, *Horcrux* covered 20.29%-146.21% more lines of code in conflict resolution logic compared to other tools. This suggests that *Horcrux* triggers consistency maintenance logic that other tools cannot, increasing the likelihood of exposing metadata inconsistencies. The results imply that the duration of inconsistency is effective for exploring conflict resolution logic.

7.3 Horcrux's Guidance Effectiveness

To check whether the inconsistency duration guidance of *Horcrux* is effective, we implemented $Horcrux_{cov}$ and $Horcrux_{ran}$, which generate cross-node file operations under code coverage guidance and without any guidance. To implement the $Horcrux_{cov}$, we collected the code coverage right after the executing of the cross-node file operations. And if the code coverage is increased, $Horcrux_{cov}$ deems the cross-node file operations as good ones. For good cross-node operations, $Horcrux_{cov}$ recognizes the first operation as the root operation and updates the relation table to add the weights for other operations. It should be mentioned that this is a heuristic approach. When $Horcrux_{cov}$ covers previously uncovered branches, we give more weights for the current operation combinations. This allows for more possibilities for further mutation based on the retained combination. However, we cannot guarantee that new mutations can cover uncovered code. As for the implementation for $Horcrux_{ran}$, we randomly update the relation table after each fuzzing round. We compared $Horcrux_{cov}$ and $Horcrux_{ran}$ with the original *Horcrux* on their effectiveness of bug detection and conflict resolution logic triggering. The results are shown in Table 7.

[2] The functions we identified are listed at: https://anonymous.4open.science/r/Horcrux-F00C/syncRelated/

Table 7: Bug numbers and conflict resolution code coverage of *Horcrux*, $Horcrux_{cov}$ and $Horcrux_{ran}$ in 24 hours.

	Horcrux		$Horcrux_{cov}$		$Horcrux_{ran}$	
	Bugs	Coverage	Bugs	Coverage	Bugs	Coverage
CephFS	3	15,918	1	14,638	0	7,501
GlusterFS	4	14,535	2	12,869	1	10,091
LeoFS	2	1,168	1	1,074	1	966
IPFS	1	1,135	0	995	0	748

The data shows that *Horcrux* found 6 and 8 more metadata inconsistencies, and covered 10.75% and 67.67% more lines of conflict resolution code than $Horcrux_{cov}$ and $Horcrux_{ran}$. We found that without the inconsistency duration guidance, it is hard to mine the relationships among cross-node file operations. Thus, $Horcrux_{cov}$ and $Horcrux_{ran}$ cannot expose the metadata inconsistencies hidden in deep conflict resolution logic. This indicates the effectiveness and necessity of the inconsistency duration guidance.

8 Discussion

8.1 Threats to the Duration Guidance

In our testing environment, the duration of inconsistencies is primarily influenced by the conflict resolution process. All nodes run within VMs on the same physical machine, eliminating significant network delays. Additionally, we disable all garbage collection (GC) pauses to prevent outliers from affecting the test results. We also control for other potential outlier effects, such as system faults, ensuring they do not interfere with the inconsistency duration feedback mechanism in *Horcrux*.

Additionally, an intuitive guidance mechanism involves using conflict resolution code coverage. In this context, a set of cross-node file operations is considered effective if it covers more lines of conflict resolution code. However, as previously discussed, achieving this is non-trivial because such code is often tightly coupled with other components. While we manually collected conflict resolution code coverage in Sections 7.2

and 7.3, identifying this code at runtime remains challenging. Therefore, for scalability, *Horcrux* is designed to be guided by inconsistency duration. To further demonstrate the equivalence of the two approaches, we manually equipped *Horcrux* with conflict resolution code coverage guidance, denoted as $Horcrux_{conflict\_cov}$, by pre-marking all conflict-related code. The results in Table 8 show that the number of bugs found by *Horcrux* under both guidance mechanisms is identical, with the conflict code coverage varying by only +/-4%.

Table 8: Bug numbers and conflict resolution code coverage of *Horcrux* and $Horcrux_{conflict\_cov}$ in 24 hours.

	Horcrux		$Horcrux_{conflict\_cov}$	
	Bugs	Coverage	Bugs	Coverage
CephFS	3	15,918	3	15,201
GlusterFS	4	14,535	4	14,492
LeoFS	2	1,168	2	1,157
IPFS	1	1,135	1	1,090

8.2 Scalability on Bug Types

Horcrux is highly scalable in terms of the vulnerability types it can detect. The existing *Horcrux* framework defines two oracles including topology oracle and property oracle, which enable the detection of metadata inconsistencies. Furthermore, through a refined design of these oracles, *Horcrux* can be effectively tailored to accommodate a wider spectrum of error scenarios in distributed file systems.

For example, the imbalance issue stands as a prevalent performance vulnerability in DFSes [11], where there is an uneven distribution of data or workloads across the nodes or storage devices within the system. For example, an imbalance issue [59] in GlusterFS can result in severe consequences such as diminished performance, resource wastage, and compromised system reliability. *Horcrux* can be adapted to detect the imbalance issue with enriched oracles that check the resource distribution among nodes. Additionally, adapting the synchronization duration calculation strategy may help guide *Horcrux* to prompt the rebalance procedure and trigger the error-prone situations.

9 Related Work

File System Testing. For distributed file system testing, Monarch [43] is the only fuzzer that generates syscalls and system faults. Similarly, the patent [12] describes a testing method to inject faults to the file blocks in DFSes to detect file corruption. Another tool smallfile [1] produces distributed file workloads on various hosts. These tools leverage a holistic infrastructure to feed inputs to DFSes. Besides, fuzzing has established itself as a widely employed technique for testing traditional file systems. Furthermore, some open-source testing suites like Teuthology [8] for CephFS and gluster-performance-test-suite [20] for GlusterFS focus on data-intensive workloads for performance and functional testing. They generate file I/Os to test the limitation of the DFSes.

Several unsupervised kernel fuzzing frameworks [26, 38, 39, 55, 56, 58] are designed to generate system calls based on predefined grammar rules. These frameworks can be effectively employed in the assessment of file systems. Additionally, there exist specialized fuzzers tailored for file system testing. Hydra [32] stands out as an extensible fuzzing framework specifically engineered to detect file system semantic bugs. B3 [45] focuses on finding crash-consistency bugs through bounded black-box crash testing. And a recent work SnapCC [40] tests the consistency of the traditional file system by exploring systematic states.

Distributed System Fuzzing. Fuzzing is also effective in uncovering vulnerabilities in distributed systems. Notably, CrashFuzz [18] employs a coverage-guided fault injection approach to test crash recovery behaviors, exposing crash recovery bugs in distributed cloud systems. Mallory [44] is a grey box fuzzer for distributed systems that applies timeline-driven testing for adaptive fault injection. Another tool named Chronos [10] leverages deep-priority fuzzing to inject transient delays to distributed systems and tries to detect bugs in timeout handling logic. Besides, TaxDC [35] contributes by creating a comprehensive taxonomy of non-deterministic concurrency bugs in distributed systems.

Main Difference. Different from Monarch, *Horcrux* focuses mainly on metadata inconsistencies by covering conflict resolution logic and mining relationships among cross-node file operations. The method in the patent [12] and Mallory [44] inject faults rather than generates inter-related cross-node file operations like *Horcrux* for inconsistencies detection. Similarly, open-source testing suites do not target interrelated cross-node file operations either. Healer [58] also mine the relationships among testing inputs. However, it learns the sequential relationships between syscalls on one node, rather than the distributed nodes. Similarly, Hydra [32] generates sequential syscalls regardless the distributed features. While TaxDC schedules the threads in the system, which is irrelevant to inconsistency triggering.

10 Conclusion

In this paper, we conducted a thorough analysis of metadata inconsistencies over the past five years in four DFSes. Based on our findings, we proposed *Horcrux*, a fuzzing framework designed for detecting metadata inconsistency bugs in DFSes. *Horcrux* maintains a file operation relation table to capture the relationships between various operations. Guided by inconsistency duration, *Horcrux* updates the relation table and generates cross-node file operations accordingly. *Horcrux* defines two monitors to detect metadata inconsistencies. We implemented *Horcrux* on four widely-used DFSes and successfully identified 10 previously unknown metadata inconsistencies.

Acknowledgment

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. U2441238, 62021002), and China Postdoctoral Science Foundation (2024M761690, GZC20240828).

11 Ethics Considerations

In conducting our research, we carefully considered the ethical implications of identifying and addressing vulnerabilities within Distributed File Systems (DFS). All discovered vulnerabilities were promptly reported to the respective developers, allowing for timely remediation before any potential exploitation could occur. This proactive approach ensured that our research contributed positively to the security and stability of the DFS environments we studied, without posing risks to users or stakeholders relying on these systems.

Furthermore, all experiments and tests were conducted within a controlled environment using our own private test cluster. We did not perform fuzz testing or any other potentially disruptive activities on live, production DFS systems. This deliberate choice was made to avoid any unintended negative impact on operational services or their users, ensuring that our research did not introduce any risks or ethical concerns related to testing in real-world environments.

12 Compliance with Open Science Policy

We fully support the open science policy. In alignment with this policy, we will openly share all relevant research artifacts associated with our work, including datasets, scripts, binaries, and source code. [3] We acknowledge the importance of this initiative in fostering transparency and collaboration within the research community and are willing to participate in the artifact evaluation process as required.

References

[1] Distributed System Analysis. distributed metadata-intensive workload generator for posix-like filesystems. https://github.com/distributed-system-analysis/smallfile, 2023. Accessed at March 27, 2024.

[2] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[3] Matt Benjamin. multisite: metadata sync does not sync sts metadata (e.g., roles, policy). https://tracker.ceph.com/issues/51068, 2021. Accessed at March 27, 2024.

[4] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

[5] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[6] Sebastian Burckhardt et al. Principles of eventual consistency. *Foundations and Trends® in Programming Languages*, 1(1-2):1–150, 2014.

[7] Ceph. Red hat ceph storage. https://www.redhat.com/en/resources/ceph-storage-datasheet, 2024. Accessed at December 16, 2024.

[8] Ceph. Teuthology – the ceph integration test framework. https://docs.ceph.com/projects/teuthology/en/latest/README.html, 2024. Accessed at December 14, 2024.

[9] Xiaoxi Chen. client: dir->dentries inconsistent, both newname and oldname points to same inode, mv complains "are the same file". https://tracker.ceph.com/issues/49912, 2023. Accessed at October 23, 2023.

[10] Yuanliang Chen. Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 109–109. IEEE Computer Society, 2024.

[11] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Zhen Yan, Qing Liao, and Yu Jiang. Themis: Finding imbalance failures in distributed file systems via a load variance model. *EuroSys 2025*, 2025.

[12] James C Davis, Willard A Davis, and Felipe Knop. Detection of file corruption in a distributed file system, March 12 2019. US Patent 10,229,121.

[13] CephFS Doc. Mount cephfs using fuse. https://docs.ceph.com/en/quincy/cephfs/mount-using-fuse/, 2023. Accessed at October 23, 2023.

[14] CephFS Doc. Mount cephfs using kernel driver. https://docs.ceph.com/en/quincy/cephfs/mount-using-kernel-driver/, 2023. Accessed at October 23, 2023.

[15] Gcov documentation. A test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2023. Accessed at March 27, 2024.

[16] Michael Eddington. protocol-fuzzer-ce. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce, 2021. Accessed at March 27, 2024.

[3] The artifacts are at: https://zenodo.org/records/14722605

[17] Greg Farnum. Mds is inconsistent about whether layouts are allowed on the root directory. https://tracker.ceph.com/issues/4220, 2023. Accessed at October 29, 2023.

[18] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. Coverage guided fault injection for cloud systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2023.

[19] gluster. Gluster filesystem : Build your distributed storage in minutes. https://github.com/gluster/glusterfs, 2023. Accessed at October 23, 2023.

[20] gluster. Setup and run perfromance test on glusterfs. https://github.com/gluster/gluster-performance-test-suite, 2024. Accessed at December 14, 2024.

[21] gluster ant. Close to open consistency fails with glsuterfs. https://github.com/gluster/glusterfs/issues/2845, 2020. Accessed at July 27, 2024.

[22] gluster ant. glustershd coredump generated. https://github.com/gluster/glusterfs/issues/919, 2020. Accessed at March 27, 2024.

[23] gluster ant. Wrong permissions syned to remote brick when using halo replication with favourite-child-policy. https://github.com/gluster/glusterfs/issues/1546, 2020. Accessed at March 27, 2024.

[24] gluster ant. glustershd coredump generated. https://github.com/gluster/glusterfs/issues/3677, 2022. Accessed at July 27, 2024.

[25] Google. american fuzzy lop. https://github.com/google/AFL, 2023. Accessed at October 23, 2023.

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

[27] GRNET. This is grnet's cloud service, for the greek research and academic community. https://okeanos.grnet.gr/home/, 2023. Accessed at March 27, 2024.

[28] HDInsight. Companies currently using ceph. https://discovery.hgdata.com/product/ceph, 2024. Accessed at December 16, 2024.

[29] HDInsight. Companies currently using glusterfs. https://discovery.hgdata.com/product/glusterfs, 2024. Accessed at December 17, 2024.

[30] ipfs. Ipfs cluster v1.0.7 is a maintenance release. https://github.com/ipfs-cluster/ipfs-cluster/releases/tag/v1.0.7, 2023. Accessed at March 27, 2024.

[31] Mohammed Rafi KC. afr/heal: Implement outcast logic in metadata heal. https://review.gluster.org/#/c/glusterfs/+/25068/, 2021. Accessed at March 27, 2024.

[32] 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.

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[34] kosfar. Surviving a ceph cluster outage: the hard way. https://blog.noc.grnet.gr/2016/10/18/surviving-a-ceph-cluster-outage-the-hard-way/, 2023. Accessed at March 27, 2024.

[35] 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 twenty-first international conference on architectural support for programming languages and operating systems*, pages 517–530, 2016.

[36] LeoProject. Leofs - a storage system for a data lake and the web. https://github.com/leo-project/leofs, 2023. Accessed at November 20, 2023.

[37] Xiubo Li. mds: the parent dir's mtime will be overwrote by update cap request when making dirs. https://tracker.ceph.com/issues/61584, 2023. Accessed at October 29, 2023.

[38] Jianzhong Liu, Yuheng Shen, Yiru Xu, and Yu Jiang. Leveraging binary coverage for effective generation guidance in kernel fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3763–3777, 2024.

[39] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–25, 2023.

[40] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Snapcc: Effective file system consistency testing using systematic state exploration. *ACM Transactions on Software Engineering and Methodology*, pages 1–23, 2025.

[41] LLVM. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2023. Accessed at October 23, 2023.

[42] Tao Lyu. Inconsistent file mode across two clients. https://tracker.ceph.com/issues/63906, 2024. Accessed at March 27, 2024.

[43] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543, Santa Clara, CA, July 2024. USENIX Association.

[44] 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, page 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

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

[47] NewDund. Grandparent objects can be removed unintentionally. https://github.com/leo-project/leofs/issues/1017, 2018. Accessed at March 27, 2024.

[48] OpenStack. The most widely deployed open source cloud software in the world. https://www.openstack.org/, 2024. Accessed at December 14, 2024.

[49] orangefs. The orangefs project. https://www.orangefs.org/, 2024. Accessed at December 14, 2024.

[50] OrbitDB. Orbitdb is a serverless, distributed, peer-to-peer database. https://github.com/orbitdb/orbitdb, 2023. Accessed at March 27, 2024.

[51] p l. Snmp don't return leo_per_object_queue. https://github.com/leo-project/leofs/issues/1123, 2017. Accessed at March 27, 2024.

[52] Qiniu. A comprehensive coverage testing system for the go programming language. https://github.com/qiniu/goc, 2023. Accessed at March 27, 2024.

[53] Rakuten. Rakuten empowers individuals and businesses around the world through innovation in internet services. https://global.rakuten.com/corp/, 2023. Accessed at November 21, 2023.

[54] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024.

[55] Yuheng Shen, Jianzhong Liu, Yiru Xu, Hao Sun, Mingzhe Wang, Nan Guan, Heyuan Shi, and Yu Jiang. Enhancing ros system fuzzing through callback tracing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–87, 2024.

[56] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4563–4574, 2022.

[57] SpaceX. Spacex. https://www.spacex.com/, 2024. Accessed at December 16, 2024.

[58] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 344–358, New York, NY, USA, 2021. Association for Computing Machinery.

[59] vnosov sf. Failure to rebalance files during execution of glusterfs command "remove-brick ... start. https://github.com/gluster/glusterfs/issues/2350, 2023. Accessed at March 27, 2024.

[60] vstax. Objects created when storage node was down don't arrive there after it's up. https://github.com/leo-project/leofs/issues/702, 2017. Accessed at March 27, 2024.

[61] 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.

[62] Wikipedia. Ext3. https://en.wikipedia.org/wiki/Ext3, 2021. Accessed at March 27, 2024.

[63] Wikipedia. The portable operating system interface. https://en.wikipedia.org/wiki/POSIX, 2023. Accessed at March 27, 2024.

[64] Wikipedia. Backus–naur form. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form, 2024. Accessed at December 14, 2024.

[65] 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, 2023. Accessed at March 27, 2024.

[66] ZenGround0. Cluster consistently fails to propogate pins to newly added peers. https://github.com/ipfs-cluster/ipfs-cluster/issues/112, 2017. Accessed at March 27, 2024.

Appendix

13 *Horcrux* under Various Cluster Scales

To test whether *Horcrux* is effective under a large-scale network that is closer to the real-world scenario, we calculate the coverage of *Horcrux* with 5, 10, 20 clients and 10, 20, 50 server nodes. The results are shown in the Table 9.

Table 9: Bug numbers and code coverage under different node scale in GlusterFS.

Node Scale	Clients(Cluster Nodes=10)			Cluster Nodes(Client=5)		
	5	10	20	10	20	50
Coverage	38,947	39,968	39,149	38,998	38,473	39,563
Bugs	4	4	4	4	4	4

We can find that the coverage does not vary much under different scales of clients (with only a variance of 1.5%), and so as the cluster nodes (with only a variance of 1.4%). Besides, there are no new bugs found when the scale of the clients or cluster nodes increases. This indicates that the cluster size does not appear to affect the performance of *Horcrux*. The consistent coverage observed across varying client scales can be attributed to the robustness of *Horcrux*'s metadata synchronization duration guidance. This guidance effectively triggers synchronization failure scenarios, irrespective of the number of cross-node operations. Besides, the stability of coverage across different scales of clusters is because the increase of cluster nodes does not trigger new logic of the consistency maintenance process. The triggering of conflict resolution depends on the workloads generated by *Horcrux*, not the size of the cluster.

14 Adaption to other DFSes

Horcrux can be adapted to other DFSes that manage metadata in a distributed manner. For DFSes that support FUSE, *Horcrux* can be directly adapted because it generates POSIX file operations. For DFSes that do not support FUSE (such as IPFS), *Horcrux* models the file commands of the DFS by writing a JSON file that records each command's operator and operands. Figure 9 shows an example of the file operations json file in IPFS.

The json file defines the IPFS command, and each command has a name and a description to describe the information of such command. Besides, each command also has a weight to show the probability to be chosen in each fuzzing round. The argument_num and the arguments items in the command describe its operands. And the last item 'pre_command' defines the command that must exist before the current command. For example, the 'delete' command must have a 'create' command before, because it has to make sure that there existing some files to be deleted.

```
{"commands": [
  {
    "name": "create",
    "description":"create a file",
    "weight": 1.0,
    "argument_num" : 1,
    "arguments":["NEW_FILE_NAME"],
    "pre_command": []
  },
{
    "name": "delete",
    "description":"delete a file",
    "weight": 1.0,
    "argument_num" : 1,
    "arguments":["EXISTING_FILE_NAME"],
    "pre_command": ["create"]
},
```

```
  {
    "name": "pinadd",
    "description":"pin add a file",
    "weight": 1.0,
    "argument_num" : 1,
    "arguments":["EXISTING_FILE_NAME"],
    "pre_command": ["create"]
  },
  {
    "name": "ipfsgc",
    "description":"gc files",
    "weight": 1.0,
    "argument_num" : 0,
    "arguments":["EXISTING_FILE_NAME"],
    "pre_command": ["create"]
  }
]}
```

Figure 9: A json file example for IPFS. To adapt Horcrux to other DFSes without the support of FUSE.

15 System Precision

According to our empirical study, metadata inconsistencies are not affected by the garbage collection process. So, no false positives are imported when we disable it. As for false negatives, among the 44 historical metadata inconsistencies, *Horcrux* found 31, indicating a false negative rate of 29.5%. The false negatives are due to two reasons: the first is that *Horcrux* cannot find inconsistencies related to data races. The other is that some bugs are related to specific requests beyond file operations. Indeed, the metric may hide metadata errors because that the metric only guides the fuzzing process to the conflict resolution logic. But all the cross-node file operations generated by *Horcrux* can trigger the local metadata update, metadata propagation, and the metadata commit phases. Thus, metadata inconsistencies in these phases can also be triggered. For example, bug#2 found by *Horcrux* occurs in the local metadata update phase.

Besides, delta debugging can be used to minimize the file operations by the following aspects: 1) minimize the cross-node file operation sequences. Most of the bugs found by *Horcrux* can be triggered by one set of cross-node file operations. To locate the precise operation set, we can use delta debugging to decrease the sequence length and find the exact operation set. 2) Not all the operations in the cross-node file operation contribute to the bug triggering. Using delta debugging may help locate which two or three parallel operations can lead to the bug.

16 System Throughput

We also evaluated the throughput of *Horcrux* on the four DFS systems. We ran *Horcrux* on each DFS for 24 hour and calculated the average IOPS, which stands for the file I/O per second. The results are shown in the following table. As the

	CephFS	GlusterFS	LeoFS	IPFS
IOPS	2.3	5.2	4.6	2.5

Table 10: The throughput of Horcrux for each DFS. The unit is IOPS, which stands for file I/O per second.

table shows, *Horcrux* sends 2.3-5.2 file operations per second. The main overhead that affect the throughput is the relation table updating phase and the cross-node file operations generation phase. Besides, *Horcrux* needs to wait for the execution to be finished before entering the next fuzzing round, which is also time-consuming. We will try to further improve the testing efficiency for *Horcrux* by optimizing these phases.

17 Historical Bug Case

We utilized a historical bug, bug#49912 in CephFS [9],to demonstrate how *Horcrux* identified it. This bug can only be reproduced by *Horcrux*, and other tools failed to reproduce it. The conditions to trigger this bug are illustrated in Figure 10.

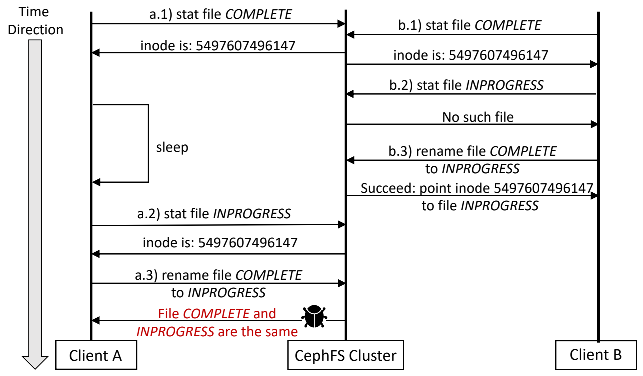

Time Direction
a.1) stat file COMPLETE
b.1) stat file COMPLETE
inode is: 5497607496147
inode is: 5497607496147
b.2) stat file INPROGRESS
No such file
sleep
b.3) rename file COMPLETE to INPROGRESS
Succeed: point inode 5497607496147 to file INPROGRESS
a.2) stat file INPROGRESS
inode is: 5497607496147
a.3) rename file COMPLETE to INPROGRESS
File COMPLETE and INPROGRESS are the same
Client A
CephFS Cluster
Client B

Figure 10: An inconsistency in CephFS during the lease/cap conflict handling process. One inode points to two file names.

The application that triggers this bug uses files in CephFS as a form of locking mechanism. Multiple threads use a file named 'INPROGRESS/COMPLETE' to determine if the lock has been acquired or released. Thus, plenty of 'mv' operations are conducted by these threads to change the file's name between 'INPROGRESS' and 'COMPLETE'. In CephFS, when handling a 'mv' operation, it first checks the state of both the source file and the destination file. As Figure 10 shows, Client A and Client B send the operation 'mv COMPLETE INPROGRESS' in parallel. Client A first checks the state of file 'COMPLETE', and the CephFS cluster returns the inode of that file. Then, Client B checks the states of both files and successfully renames the file from 'COMPLETE' to 'INPROGRESS'. After that, Client A continues checking the state. However, the 'mv' operation from Client B has already been committed and the inode '549760749614' has pointed to the file 'INPROGRESS'. As a result, the CephFS tells Client A that both files 'COMPLETE' and 'INPROGRESS' exist, and they are the same file.