
# When Amnesia Strikes: Understanding and Reproducing Data Loss Bugs with Fault Injection

Maria Ramos*
INESC TEC & U. Minho
maria.j.ramos@inesctec.pt

João Azevedo*
INESC TEC
joao.azevedo@inesctec.pt

Kyle Kingsbury
Jepsen
aphyr@jepsen.io

José Pereira
INESC TEC & U. Minho
jop@di.uminho.pt

Tânia Esteves
INESC TEC & U. Minho
tania.c.araujo@inesctec.pt

Ricardo Macedo
INESC TEC & U. Minho
ricardo.g.macedo@inesctec.pt

João Paulo
INESC TEC & U. Minho
joao.t.paulo@inesctec.pt

## ABSTRACT

We present LazyFS, a new fault injection tool that simplifies the debugging and reproduction of complex data *durability* bugs experienced by databases, key-value stores, and other data-centric systems in crashes. Our tool simulates persistence properties of POSIX file systems (*e.g.*, operations *ordering* and *atomicity*) and enables users to inject `lost` and `torn` write faults with a precise and controlled approach. Further, it provides profiling information about the system's operations flow and persisted data, enabling users to better understand the root cause of errors.

We use LazyFS to study seven important systems: PostgreSQL, etcd, Zookeeper, Redis, LevelDB, PebblesDB, and Lightning Network. Our fault injection campaign shows that LazyFS automates and facilitates the reproduction of five known bug reports containing manual and complex reproducibility steps. Further, it aids in understanding and reproducing seven ambiguous bugs reported by users. Finally, LazyFS is used to find eight new bugs, which lead to data loss, corruption, and unavailability.

**PVLDB Reference Format:**
Maria Ramos, João Azevedo, Kyle Kingsbury, José Pereira, Tânia Esteves, Ricardo Macedo, and João Paulo. When Amnesia Strikes: Understanding and Reproducing Data Loss Bugs with Fault Injection. PVLDB, 17(11): 3017 - 3030, 2024.
doi:10.14778/3681954.3681980

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/dsrhaslab/lazyfs.

## 1 INTRODUCTION

If no special care is taken in the presence of faults (*e.g.*, power outages and crashes), modern storage solutions cannot ensure that all pending write operations are actually finished and can, in fact, complete only an arbitrary subset of them [1, 2, 29, 33, 36, 53]. This is a major issue for databases, key-value stores, and other data-centric systems, as naive implementations might result in lost data, in relation to what has already been acknowledged to users, or even in outright data corruption, as the internal consistency of persistent structures is endangered. Therefore, many of these systems use transactional techniques such as the classic DO-REDO-UNDO protocol to avoid loss and corruption of data [14]. The system judiciously keeps some redundancy in a log file while modifying persistent data structures such that it becomes possible, after a fault, to roll back unfinished operations and replay finished ones that have not been fully applied, regardless of the completion status of writes at the time of the fault.

Thus, these fault-tolerant techniques critically depend on the ability to order operations that write to persistent storage and wait for their completion. Within POSIX Operating Systems (OS), this can ostensibly be achieved through some variation of the `fsync` system call, which ensures that data previously written and cached at different layers of the stack (*e.g.,* file system, block device) is actually flushed to the underlying storage device [44, 50]. This leads to a dilemma for system developers: to easily ensure correctness, one should write small portions of data, handled atomically by the storage stack, and call `fsync` to ensure their *durability* and *order*. However, this greatly restricts parallel I/O and underuses available bandwidth, with a profound impact on performance [5, 53].

For performance reasons, many systems issue multiple `write` calls or single ones with larger payloads (*i.e.*, size of the content being written) before explicitly flushing written data. This greatly increases the variety and complexity of possible failure scenarios. For example, writes may be completely lost (*i.e.*, `lost` writes) or may be persisted partially at the storage device (*i.e.*, `torn` writes). Data may even be persisted out of temporal order, *i.e.*, more recent data is persisted, but older data is lost [1, 2, 5, 36]. This may happen because the storage stack has several layers where `write` calls may be reordered and flushed in background to the underlying device. As an example, consider that a system issues *write1* followed by *write2*, but the latter is reordered at the file system's layer and flushed first. If an OS crash happens before flushing *write1*, then only the content of the second write is persisted. The same can happen when writing payloads that are larger than the page size of the file system's cache. For example, consider the system issues a `write` call with a payload of 8 KiB, which is divided into two 4 KiB pages at the file system layer. If the first page is flushed but a crash happens before flushing the second, the latter's content will not be available upon recovery.

The core challenge for developers is that there is a strong incentive to reduce the use of `fsync` for performance, but this leads

*Both authors contributed equally to this research.


Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681980

to increased complexity in developing, maintaining, and testing systems [37, 47]. Even in mature systems, it becomes hard to correctly identify and reproduce crash consistency problems. As the first contribution of this paper, we validate this challenge by studying twelve bug reports from widely used database, key-value, coordination and blockchain systems. Our findings (§2.2) show:

- Many reports are ambiguous and there is no clear association between the error reported by the system and the fault that originated it.
- When available, the steps to reproduce bugs include manual (*e.g.*, plugging of the server's power cord) and complex (*e.g.*, changing source code) instructions.
- Current tracing and fault exploration tools are helpful but insufficient to quickly identify, reproduce, and validate the bugs and potential fixes for these.

Therefore, system developers need a better approach to simplify and automate the identification, reproduction, and validation of these bugs. As our second contribution, we propose LazyFS, a software-based tool for injecting `lost` and `torn` write faults at the file system level, and aimed precisely at testing software solutions with strong data *durability* requirements. Its main goal is to aid developers in reproducing bugs and validating the crash consistency of their systems while providing insightful information about the cause and impact of observed errors.

Briefly, LazyFS implements its own in-memory page cache, which provides complete control over when written data is flushed into persistent storage. Consequently, it can generate scenarios that are possible but hard to reproduce by enabling developers to specify the type, timing, and placement of faults to be injected, either in a static configuration file or at runtime, through a custom API. Our tool allows testing any POSIX-compliant system and does not require any code changes. Moreover, LazyFS provides insightful profiling information about the flow of system calls and state of written data (*i.e.*, persisted and cached data) so that developers can better understand the root cause of errors.

As the third contribution, we use LazyFS to study seven important systems: PostgreSQL [43], etcd [12], Zookeeper [15], Redis [46], LevelDB [25], PebblesDB [35], and Lightning Network [27]. Our fault injection campaign shows that LazyFS eases and automates the reproduction of twelve known bug reports either containing manual and complex steps, or partial and ambiguous information, for their reproducibility. Further, we identify eight new bugs, from which four are already confirmed by the developers [8–10]. One of them is discovered through the integration of our tool with the Jepsen distributed systems testing framework [16]. Currently, LazyFS is integrated into the testing environments of two production-level systems, namely etcd and MongoDB [11, 30].

## 2 BACKGROUND AND MOTIVATION

The typical flow of write operations from systems using a kernel-based POSIX storage stack is shown in Figure 1. `Write` system calls, or related ones (*e.g.*, `pwrite`, `writev`), are intercepted first at the File System layer (1), then go through the Block I/O layer (2) and finally reach the Disk hardware layer (3).

To improve write performance, each layer usually buffers data in volatile memory (*e.g.*, in the file system's page cache or block layer's I/O cache) [1]. Buffered data is then flushed asynchronously to the underlying layer by the operating system to better leverage the available I/O bandwidth. Further, layers may implement other performance optimizations, such as I/O scheduling and batching [44]. The latter may change the temporal order in which the contents of consecutive `write` calls, or even of a single large call (*e.g.*, when it spawns across multiple file system pages), are forwarded into the next layers and persisted at the disk device.

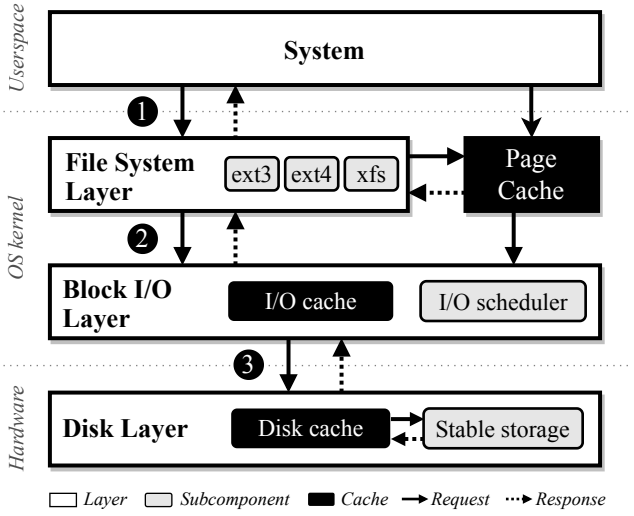


**Figure 1: I/O flow of kernel-based POSIX storage stacks.**

Although these designs offer better performance, they also increase the probability of data loss under failures (*e.g.*, a server's power outage). Figure 2 depicts three data loss scenarios that may happen. For simplicity, and as an example, let us assume that *A*, *B*, *C*, and *D* are 2 KiB fixed-size blocks that are grouped into 4 KiB pages at the *Cache* layer, a common size for many file system implementations [4, 28, 49]. Note that these faults may happen if blocks are written by a sequence of multiple small `write` calls or by a single large `write`.

**Lost write(s).** One possible outcome is that all written data (*ABCD*) is lost, potentially breaking the data *durability* requirements of the system. This may happen because, at the moment of the crash, the OS may not have yet triggered the flush of this content to disk.

**Linear torn write(s).** Alternatively, *AB* may be flushed asynchronously to the device, but *CD* is lost. After recovery, the system will only have access to partial data on the disk. Besides breaking data *durability*, this fault may also break *atomicity* assumptions made by developers. For instance, the system might assume that a sequence of consecutive `writes`, or a single large `write` call, are either fully persisted or lost under failures.

**Non-linear torn write(s).** Another outcome is that *CD* is flushed asynchronously to the device, but *AB* is lost. After recovery, the system will only have access to partial data on the disk, potentially breaking *durability* and *atomicity* assumptions. This fault may break the temporal *ordering* of the operations—for instance, if the system wrote *AB* before *CD*. This can happen because `write` calls, or the

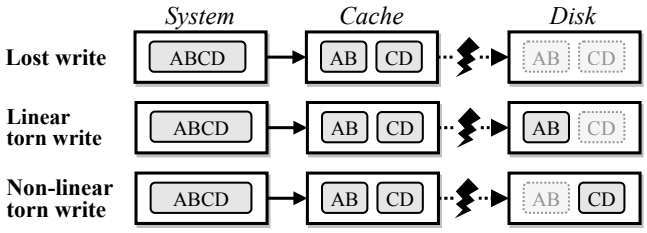


**Figure 2: Possible `lost` and `torn` write scenarios under faults.**

pages where these are buffered, may be rescheduled and flushed in a different order by the I/O layers (*e.g.*, file system) [36].

The question, therefore, is: how can developers control these asynchronous behaviors and make crash-consistent systems? In kernel-based POSIX storage stacks, this can ostensibly be done with the `fsync` system call or related ones (*e.g.*, `fdatasync`). The `fsync` call ensures that all modified cache pages for a given file are transferred to disk to make the information available even if the system crashes or is rebooted [44, 50]. There are also high-level patterns for atomic writes. For instance, developers may `write` to a temporary file, `fsync` it, and `rename` it to the final file [37, 53].

The caveat is that developers must use these calls wisely. If used for every disk sector (*e.g.*, 512 bytes) of written data, or even for every block aligned with the file system's page size (*e.g.*, 4 KiB), it adds impractical overhead to the systems' I/O performance. If used less frequently, *i.e.,* after a sequence of `writes` and/or a large `write`, one gets better performance. However, upon a failure, data *durability*, *atomicity* and *ordering* assumptions may yet be broken.

To highlight the impact and the complexity of identifying, reproducing, and fixing these faults, we next analyze twelve bugs found in six widely used systems. We examine how users report such bugs and answer the following questions:

- What are the symptoms reported by users?
- What is the impact on the system?
- How are the issues reproduced and fixed?

We selected bugs from one relational database (PostgreSQL [43]), three key-value stores (LevelDB [25], etcd [12] and Redis [46]), one distributed coordination system (ZooKeeper [15]) and one blockchain application (Lightning Network [27]). We found these bug reports by examining the systems' mailing lists, GitHub repositories, JIRA repositories, and blogs. Our aim is to study examples of bugs linked to the aforementioned faults and not to comprehensively analyze all systems exhibiting them.

## 2.1 Bugs overview

We next summarize each bug report by considering the interaction between the reporter and the development team. When available, we detail the information provided by the user to troubleshoot and reproduce the bug and the insights gained from the discussion with the development team.

***Bug #1* (2012).** A PostgreSQL user reports a corruption of the statistics file (*pgstat.stat*) after restoring the database from a cold backup. The development team suggests this corruption can result from a partially written statistics file in the backup [39].

***Bug #2* (2013).** A ZooKeeper user mentions an error on the transaction log. The error, which occurs after a crash on one of the servers, indicates that the log file has an invalid magic number [58].

***Bug #3* (2014).** A LevelDB user reports a bug that occurs when a power crash happens while a database is being created. The user provides reproduction steps, highlighting that it is only triggered if the crash occurs within a narrow time window and with specific file systems. As stated, the bug results in an I/O error from LevelDB and its possible cause is a missing `fsync` call after creating the *MANIFEST-000001* file [24].

***Bug #4* (2014).** A LevelDB user reports data corruption and consequent failure in the database recovery. To reproduce the bug, the user suggests turning off the PC while writing a big data block [21].

***Bug #5* (2014).** A LevelDB user describes potential data corruption resulting from missing `fsync` calls to rotated commit log files. Namely, given the right timing and if a power loss happens, data from an older log is lost while data from a more recent log is not [23].

***Bug #6* (2014).** A LevelDB user describes a bug that leads to corrupted values. LevelDB appends the data from `Put` requests to the commit log file. If a power loss happens, the last append can end up with garbage or zeros. When users try to retrieve the values from the database, corrupted values are returned [22].

***Bug #7* (2015).** A ZooKeeper user reports an error at the initialization phase caused by an empty transaction log. The user explains that the problem results from killing the ZooKeeper server after the creation of a new log file and before its header is flushed [59].

***Bug #8* (2016).** A ZooKeeper user reports a potential cluster unavailability scenario. When ZooKeeper appends data to the transaction log file, it issues four write operations before the `fdatasync` call. If a crash happens before the `fdatasync` call, the file system may persist the fourth `write` and fail to persist the third. In this scenario, ZooKeeper fails to start with a checksum mismatch error [60].

***Bug #9* (2016).** An etcd user reports that etcd data becomes corrupted with a checksum mismatch error, and the only recourse is to discard the current data directory and start afresh. The comments explain that the issue results from a `torn` write [6].

***Bug #10* (2019).** Another etcd user reports a similar issue to *Bug #9*, with etcd v3 throwing the same error about CRC mismatch after a hard reboot. The author states the bug occurs due to a bad disk, causing the file system to fail and lose writes [7].

***Bug #11* (2022).** A Redis user suggests saving the cluster config and access control list files safely with an atomic pattern (*i.e.*, write to a temporary file, sync the file, rename it to the final name, and sync the directory) [45].

***Bug #12* (2022).** A Lightning Network user notes that the function `ioutil.WriteFile` does not synchronize the data to disk, and thus, any data written with it can be totally or partially lost. Even though the user cannot evaluate whether any of this function's invocations are critical, they suggest issuing `fsync` calls for a safer solution [31].

## 2.2 Takeaways

Table 1 categorizes each bug based on the reported impact, reproduction steps, and current status. We also identify the faults associated with each bug and the affected files. Next, we provide

**Table 1: Bugs classification according to their fault type, impact, affected files, reproduction steps and current status.**

| Bug | System | Fault Type | Impact | | | Affected Files | Reproduction Steps | | | | | Issue Status |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *Data loss* | *Corruption* | *Unavailability* | | *Reproducible* | *Code changes* | *Specific setup* | *Unplug machine* | *Time sensitive* | |
| **#1** | *P* | *L* | ○ | ● | ○ | *O* | ● | ○ | ● | ○ | ○ | – |
| **#2** | *Z* | *T* | ○ | ○ | ● | *L* | ○ | – | – | – | – | – |
| **#3** | *L* | *L* | ○ | ○ | ● | *M* | ● | ● | ● | ● | ● | ○ |
| **#4** | *L* | *L* | ○ | ● | ● | *M* | ● | ○ | ○ | ● | ● | ● |
| **#5** | *L* | *L* | ● | ○ | ○ | *L* | ● | ● | ● | ● | ● | ○ |
| **#6** | *L* | *T* | ○ | ● | ○ | *L* | ● | ● | ● | ● | ● | ○ |
| **#7** | *Z* | *L* | ○ | ○ | ● | *L* | ○ | – | – | – | – | ○ |
| **#8** | *Z* | *T* | ○ | ○ | ● | *L* | ○ | – | – | – | – | ○ |
| **#9** | *E* | *T* | ○ | ● | ● | *L* | ○ | – | – | – | – | ● |
| **#10** | *E* | *T* | ○ | ● | ● | *L* | ○ | – | – | – | – | ● |
| **#11** | *R* | *L* | ● | ○ | ○ | *O* | ○ | – | – | – | – | ● |
| **#12** | *LN* | *L* | ● | ○ | ○ | * | ○ | – | – | – | – | ○ |

**Properties**
● Reported
○ Not reported
* Unspecified
– Not Applicable

**System**
***E*** - etcd
***L*** - LevelDB
***LN*** - Lightning N.
***P*** - PostgreSQL
***R*** - Redis
***Z*** - ZooKeeper

**Fault Type**
***L*** - Lost write
***T*** - Torn write

**Files**
***L*** - Log
***M*** - Metadata
***O*** - Other

**Status**
● Closed
○ Open

general takeaways that highlight the impact of these bugs and then further discuss how developers are reproducing and fixing them.

Of the twelve bugs, three result in data loss, such as the loss of key-value pairs in LevelDB (*Bug #5*) or the loss of users' authentication rules in Redis (*Bug #11*). Five of the bugs lead to corruption of information at different granularity levels, including data corruption (*e.g.*, corrupted values in *Bug #6*), and file corruption (*e.g.*, corrupted metadata files in *Bug #4*).

Ultimately, seven of the bugs affect availability. For example, in Zookeeper, these faults make some servers unable to bootstrap, compromising the faulty server(s) and potentially the entire cluster (*e.g.*, *Bug #8*). Similarly, some bugs cause databases and key-value stores, such as LevelDB and etcd, to become inaccessible and often unrecoverable (*e.g.*, *Bugs #4* and *#9*).

Data corruption and service unavailability are severe outcomes for critical systems; one must have efficient ways to quickly reproduce and fix such bugs. Next, we show that the current process for handling bugs related to `lost` and (*linear* and *non-linear*) `torn` writes is inefficient.

*2.2.1 Reproducibility.* Some of the reports (five out of twelve) include instructions on how to reproduce the bug. However, the reproduction steps are often complex to execute, time-consuming, and require a significant amount of manual effort from users.

**Finding 1.** Many of these bugs are time-sensitive, occurring on exceptional occasions and at very narrow time windows.

Namely, four of the reports suggest ungraceful server shutdowns to simulate power crashes, such as pulling out the machine's power cord, and include precise instructions on the timing to do so. For instance, *Bug #3* indicates that *"the crash should happen before any sync-like call after rename("000002.dbtmp")"*, and *"within around few seconds"*. Such a process can easily become time-consuming, as the user may need to repeat the same process multiple times to correctly reproduce the bug.

*Takeaway*: Ideally, one would be able to reproduce a bug without requiring manual steps or having to repeat the same process multiple times.

**Finding 2.** The reproducibility of many bugs requires modifying the systems' codebases to insert small portions of code that allow the error to manifest.

The most typical case is to add `sleep` calls in strategic parts of the code (*e.g.*, after the `msync` call in *Bug #3*; before updating the *MANIFEST* file in *Bug #5*) to allow the developer to inject the fault (*e.g.*, ungraceful server shutdown) at the right time.

*Takeaway*: Ideally, the reproduction process should treat the systems as opaque boxes, allowing to reproduce the bug without requiring any codebase modifications.

**Finding 3.** Besides the narrow time window, many of these bugs only manifest in very specific and complex deployments.

Some bugs are linked to the underlying file system and its configuration modes. As a result, the reproduction process often requires a separate partition with a specific file system (*e.g.*, `ext4`, `ext3` with writeback mode). Other bugs happen only when several (nondeterministic) characteristics are met (*e.g.*, when restoring a database from a cold backup with a corrupt file, as in *Bug #1*).

*Takeaway*: One should be able to reproduce bugs without requiring tailored deployments and configurations or depending on complex preconditions (*e.g.*, corrupted files at the time of a backup).

**Finding 4.** Some reports mention possible solutions to fix the identified bugs. However, these solutions can only be effectively validated if there is a way to reproduce the erroneous behavior.

Additionally, some systems provide mechanisms for recovering from a corrupted state (*e.g.*, the `RepairDB` function from LevelDB). However, there are cases where these are ineffective (*e.g.*, *Bug #6*).

*Takeaway*: Being able to reproduce bugs also helps test and improve repair systems.

> **Summary.** *A non-intrusive and automated way to reproduce storage-level data loss bugs is currently lacking and is of utmost importance so that developers can identify and analyze problems reported by users and validate their fixes.*

*2.2.2 Understanding ambiguous bugs.* Many bug reports (seven out of twelve) are ambiguous, mentioning only the problem observed (*e.g.*, data loss, corruption, or system unavailability) and, when applicable, the error reported by the system, without providing any insights on their root cause or how to reproduce them. As a consequence, some of these reports end up being disregarded by the development team, as there is no way to reproduce the bug and, consequently, to understand the problem and fix it.

**Finding 5.** In most cases, it is hard to associate the reported error with the fault that originated it (*i.e.*, `lost` or `torn` write). Yet the type of affected files, which is a common characteristic reported by users, can aid in determining the underlying fault.

In nine of the bugs, the problem is related to log or metadata files. Namely, several reports mention problems related to log files in LevelDB (Bug #5, #6), transaction logs in ZooKeeper (Bugs #2, #7, #8), and Write-Ahead Logs (*WAL*) in etcd (Bugs #9, #10). Metadata files, such as LevelDB's *CURRENT* and *MANIFEST* files, are also mentioned in some bug reports (*e.g.*, *Bug #3* and *#4*).

*Takeaway*: Ideally, one should leverage this type of information from users, test critical files against `lost` and `torn` writes, and verify if the reported symptoms/errors are similar.

**Finding 6.** When reported information is ambiguous, developers lean on external tools to understand what may have happened.

The strace tool is often used to observe system calls submitted by the system to the kernel and help understand which system actions could have led to the bug (*e.g.*, *Bug #5*) [48]. However, strace is only helpful for debugging the problem rather than pinpointing the associated fault or reproducing the bug. Another tool is ALICE, which explores and identifies system actions that may lead to `lost` and `torn` writes (*e.g.*, *Bug #6*) [36]. Although, since it is based on system traces and file system modeling, it does not provide the means for developers to reproduce specific bugs reported by users.

*Takeaway*: A tool for reproducing `lost` and `torn` write faults is still missing. Such a tool could be combined with strace and ALICE to provide developers a better way to explore, debug, reproduce, and fix bugs.

**Finding 7.** Data loss and corruption are two common symptoms of these bugs, but both can go unnoticed by systems (and users).

Sometimes the studied faults cause silent errors. In these cases, the problem (*e.g.*, data loss or corruption) is not identified or reported by the system. Instead, the problem is only noticed when a user performs specific system actions, such as listing all key-value pairs in the database (*e.g.*, *Bugs #5* e *#6*).

*Takeaway*: When testing systems and injecting faults, one should be able to obtain relevant information about partial or total data loss, even when the system does not detect or report it.

**Finding 8.** There are similarities regarding the error messages and the affected files mentioned in bugs across different systems.

The first similarity goes back to *Finding 5*, which states that these faults typically affect log and metadata files. There are also different bugs with similar error messages. For instance, both *Bugs #9* and *#10* mention checksum errors.

*Takeaway*: One could leverage these patterns to test new and unexplored systems against known bugs and failures.

> **Summary.** *A tool that allows testing systems against lost and torn write faults and obtaining insightful information on how errors are triggered is essential for developers to understand ambiguous bugs reported by users and for users to provide more detailed information when reporting newly found bugs.*

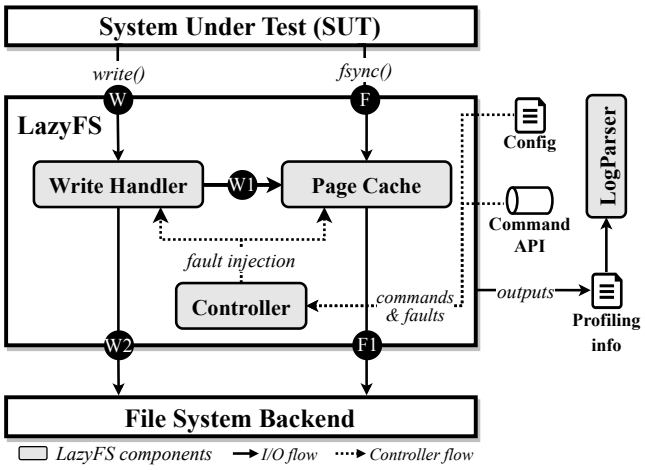


**Figure 3: LazyFS's architecture and flow of operations.**

## 3 LAZYFS

We propose LazyFS, a software-based tool for injecting `lost` and `torn` write faults at the file system level. The main goals of LazyFS are to automate the current manual and intrusive process used to reproduce bugs and aid users in testing the resiliency of their systems while providing insightful information about the cause and impact of observed errors.

### 3.1 Design goals

In compliance with the takeaways discussed in §2, LazyFS's design is built over the following core principles:

- **Closed-box solution.** Our tool is transparent to the system being tested, avoiding any modifications to its codebase. The only requirement is that the system is compatible with the POSIX interface.
- **Reproducible fault injection.** LazyFS implements a custom page cache that enables complete control of when and how data is synchronized to disk. This allows creating reproducible fault scenarios by eliminating the nondeterminism originated by the OS cache flushing mechanisms.[1]
- **Comprehensive file system behavior.** LazyFS allows mimicking different persistence properties of modern file systems, including data *durability*, *atomicity*, and temporal *ordering* of write requests.
- **Informative profiling.** Our tool helps users identify the root cause of errors by precisely pinpointing data at risk of being lost or torn under a given fault.
- **Combinable design.** LazyFS can be used directly by users or as a module of complementary testing systems to expand the range of covered faults [16].

### 3.2 Overview

LazyFS provides a FUSE-based file system that acts as an intermediary passthrough I/O layer between the System Under Test (SUT)

[1] LazyFS's goal is to assess the crash consistency of systems using the OS page cache. Solutions that bypass the OS cache (*e.g.*, use the O_DIRECT flag [32]) are out of scope.

and a regular persistent file system backend (*e.g.*, `ext4`), as depicted in Figure 3. Our solution is used as a typical POSIX file system, supporting all the necessary system calls to operate over files and directories' data and metadata.

Data written by the SUT (*e.g.*, through `write` or `pwrite` calls) (W) is stored in the internal Page Cache of LazyFS (W1). This cache follows a similar design to the one found in modern Linux file systems, with a single but important exception: it does not perform background flushes to the next I/O layer. Data is flushed to the file system backend (F1) only when the SUT explicitly issues a synchronization system call (*e.g.*, `fsync` or `fdatasync`) (F).

The ability to retain written data in the cache and control when and how to flush it is what allows LazyFS to mimic the exact behavior of `lost` and `torn` write faults. Users are responsible for specifying the type, timing, and placement of faults to be injected. According to the user's specification, LazyFS causes full or partial loss of unflushed data. By inspecting potential system errors caused by missing data at the persistent file system backend and combining these with the output information provided by LazyFS, users can explore and understand the impact and errors of their fault injection campaign.

Next, we further explain each of the main architectural components of our solution, describe the supported fault scenarios, and detail other features and implementation considerations.

## 3.3 I/O requests handling

System calls issued by the SUT are intercepted by LazyFS and handled by two main modules.

**Page Cache.** The content of all `write` calls is buffered at the Page Cache module. This cache follows the design found in modern Linux file systems [1, 3]. Namely, each file name is associated with an inode, which in turn is mapped to a list of dirty pages that contain unflushed data. Each page is divided into blocks, while the number and size of blocks per page are configurable by users to mimic different file system configurations.

Dirty pages are only flushed to the persistent file system backend when: *i)* an `fsync`-related call for the file is explicitly issued by the SUT; *ii)* faults for persisting partial data are injected; or *iii)* the Page Cache size, defined by users, is full. In the last case, we follow a Least Recently Used (LRU) eviction policy. To ensure data consistency, `read` calls are served directly from the Page Cache when dirty data is requested. As in other regular passthrough FUSE-based file systems, the metadata of each file (*e.g.*, size, permissions) is updated and persisted by the file system backend [38, 52].

**Write Handler.** The Write Handler module supports the injection of `torn` write faults. According to the fault injection campaign specified by the user, the module may operate in two distinct ways.

When one wishes to test the SUT's resiliency to a sequence of `write` calls that may be torn, the module will write the content of specific calls directly into the file system backend (*e.g.*, first and fourth calls from a group of four consecutive writes) (W2). The content written by non-targeted writes (*i.e.*, second and third calls in the previous example) is buffered at the Page Cache (W1) and is only flushed if the SUT explicitly issues an `fsync`-related call.

A slightly different approach is taken for testing systems against large `write` calls (*i.e.*, spawning across multiple pages) that may be torn. In this case, the Write Handler first splits the targeted write into $N$ parts as defined by the user. Then, the content of specific parts (*e.g.*, second and third) is written directly into the file system backend. The remaining parts are only written to the Page Cache.

## 3.4 Fault injection

LazyFS allows users to specify faults to inject, their timing, and parameters (*e.g.*, targeted file name and/or system call type) through a command-based API or a configuration file. The API allows issuing faults interactively while LazyFS and the SUT are running. These are injected asynchronously, *i.e.*, while the system is performing requests to LazyFS. Thus, the fault injection timing must be controlled, at runtime, by the user. The configuration file provides a synchronous fault injection environment where users specify, a priori, the exact timing when the fault is to be injected (*e.g.*, inject a `torn` write for the fifth `write` call to file $A$).[2]

User configurations and commands are parsed by the **Controller** component, which coordinates the injection of the following faults.

**Lost write(s).** By clearing all content buffered at LazyFS's Page Cache, the Controller mimics the full loss of data written by the SUT that was not explicitly flushed. We provide two different options for injecting `lost` writes, each of which is used in this paper to simulate scenarios where, for instance, a given server fails abruptly.

In more detail, users may issue a `crash` fault through an API command or a static configuration following the same nomenclature. The latter requires a certain precondition to inject the fault (*e.g.*, at the first `write` call issued for file $A$).

Under a `crash` fault, LazyFS is terminated, immediately leaving the SUT unable to make requests or obtain responses from the underlying file system and, eventually, leading to the SUT's failure.

Alternatively, we provide a `clear-cache` command/configuration that clears unflushed data at LazyFS's page cache but leaves the file system running. Therefore, the coordination of the SUT's crash and fault-injection timing is left to users. For frameworks that automate this coordination process, such as Jepsen, the `clear-cache` option provides the means to simulate data loss without requiring re-mounting LazyFS multiple times (Section 5.3).

**Torn write(s).** By instructing the Write Handler component, the Controller can inject both `linear` and `non-linear torn` write faults. When users want to test the resiliency of a group of writes against these faults, they must specify the group and the specific calls that should be written directly to the file system backend in order to define fault injection timing. For example, for the second group of four consecutive writes issued by the SUT, flush the first two calls to the file system backend (`linear torn` write). Or, as another example, for the first group of three consecutive writes, flush only the third (`non-linear torn` write).

Alternatively, if users want to target large writes, they need to indicate the system call to consider and which parts from its content are to be flushed directly. For example, for the fifth `write` call issued to file $A$, flush the first two parts (`linear torn` write). A `non-linear torn` write fault variant could follow the same parameters, but users would instead, for instance, configure the second and third parts to be flushed.

[2] In this example, the reproducibility of experiments still depends on the determinism of the SUT's workload.

Both types of faults forcefully terminate LazyFS and can be injected through the API or through the static configuration file (commands/configurations `torn-seq` and `torn-op`). Also, note that these may require some a priori knowledge from users to be injected successfully. Namely, how can users know if the SUT is issuing groups of consecutive writes or large writes? To aid in such a task, our tool provides additional *profiling* information to help define fault injection timing (*e.g.,* the ordinal of the write operation), as explained in the next section.

## 3.5 Output

LazyFS outputs informative details about the SUT's actions that may help identify possible points of fault injection and explain the root cause for errors under `lost` and `torn` write faults.

**Cache state.** When total or partial data loss occurs, it is essential to understand which file(s) content is effectively lost. To assist in this task, immediately before injecting a fault, LazyFS registers the file(s) blocks only residing at the Page Cache that will be lost and outputs this information to the user.

The same information (*i.e.*, file(s) blocks at risk of being lost) can be obtained through LazyFS's `unsynced-data-report` API command at runtime. This feature can be used even if no faults are being injected and is useful to understand ambiguous bugs and to find new ones. For example, by running the system on top of LazyFS and querying the latter for data at risk, the user can identify potential moments for fault injection.

**Requests trace.** Knowing what data was lost is not always sufficient to understand and find the root cause of errors. Hence, LazyFS creates a log of all system calls issued by the SUT (*e.g.*, `open`, `read`, `write`, `fsync`, `close`, `rename`), along with their arguments (*e.g.*, size, file path) and the moments of fault injection and activation. By exploring this information, users can observe the flow of operations and, for instance, identify large `write` calls, or groups of consecutive writes, that may be at risk of being torn.

To facilitate the interpretation of logged information, LazyFS provides a **LogParser** that allows: *i)* filtering system calls by a specific type or file name; *ii)* visually representing a sequence of system calls; *iii)* identifying groups of consecutive system calls with the same type; and *iii)* identifying write calls that exceed a user-configurable page size.

Figure 4 shows an example of a graph produced by LazyFS's LogParser. By analyzing the graph, one can observe the system calls made by the SUT to two different files (*i.e.*, `file1.txt` and `file2.txt`). Further, one can easily identify two groups of calls, one corresponding to a sequence of `write` and `fsync` calls repeated twice (A) and another composed of three consecutive `write` calls (B). Such information helps determine possible fault injection scenarios. In this case, we could configure a `torn` write fault to be injected upon the first group of consecutive `write` calls.

## 3.6 Implementation

LazyFS is implemented in 2,5K lines of C++17 code and uses the FUSE library (v3) [26]. It supports all POSIX system calls offered by FUSE, and therefore of a traditional kernel-based file system. In detail, our solution builds up on FUSE's *passthrough* implementation while extending the behavior of specific data (*i.e.*, `read`, `write`, `truncate`, `fsync`, `fdatasync`) and metadata (*i.e.*, `getattr`, `open`, `create`, `rename`, `link` and `unlink`) operations to support the page cache, fault injection and tracing functionalities.

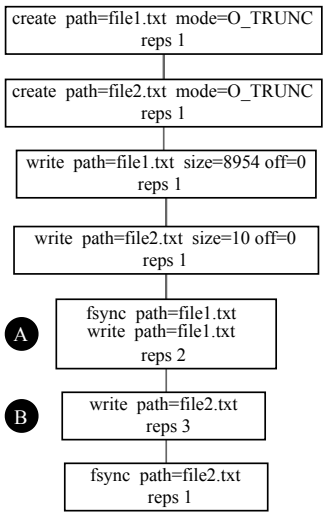


**Figure 4: Graph generated by LazyFS's LogParser. The representation shows system calls issued by the SUT, including their arguments (*e.g.*, file path, size). Consecutive calls of the same type(s) are grouped into the same rectangle. *Reps X* — indicates the number of repetitions for a group of calls.**

The static configuration file is implemented with the *TOML* language. The command API is exposed by a dedicated Unix FIFO (*i.e.*, named pipe). Commands received at the FIFO are processed asynchronously by a dedicated background thread, avoiding any contention on the main flow of I/O operations issued by the SUT and handled by LazyFS.

LazyFS's LogParser is implemented in ≈300 lines of Python3 code and leverages the *pydot* module for creating graph visualizations.

## 3.7 Usage and Integration

We now detail how our solution can be used and integrated with other testing frameworks.

**Usage.** Setting up a fault injection campaign is straightforward. First, the user launches LazyFS by mounting it as a passthrough file system on top of the persistent file system backend (*e.g.*, a `ext4` file system). Then, the user runs the SUT, which should be configured to persist data under the directory tree managed by LazyFS.

Through the static configuration file, which is specified before bootstrapping LazyFS, or by using our fault injection API at runtime, the user can specify the fault(s) to inject. After injecting the fault, the SUT may end up in different states, *i.e.*, continue running, clean shut down, or crash. In the last two cases, the user can restart the system on top of LazyFS and check for any errors.

**Integration.** LazyFS can be used directly by users or integrated within existing testing environments. To showcase the latter, we have integrated it with Jepsen, a framework to test the reliability of distributed systems [16]. Jepsen is used to evaluate databases,

coordination services, and distributed queues (among others) and was able to find and confirm multiple data consistency bugs [17].

Jepsen provides pluggable fault injection, including crashing processes, manipulating network traffic, and introducing errors in system clocks. We added three hundred lines of code to Jepsen which installs, configures, and induces faults via LazyFS, using our API for `lost` write faults. Support for `torn` write faults is deferred for future work.

## 3.8 Limitations

In §5, we validate that LazyFS can be used to understand and reproduce reported bugs and to uncover new ones. However, we acknowledge that our tool has limitations, whose solutions are out of scope for this paper but will be explored as future work.

**Exploration of new bugs.** Our solution is built with the main goal of helping users reproduce and understand known bugs. Regardless, as shown in §5.3, our solution can be used to find new errors. Nonetheless, we do not claim it to be a bug exploration tool such as ALICE [36] or Jepsen [16] that can automate bug detection. In fact, we show that these are complementary tools that can be combined to be more effective.

**Metadata fault injection.** Although LazyFS leverages file's metadata (*e.g.*, file names) for injecting faults, it does not allow assessing the durability of metadata (*e.g.*, whether updates to inodes are flushed correctly to disk). As shown in the literature, these are important faults that can benefit from a reproducible approach [2, 36].

**Workloads determinism.** When workloads are deterministic (*i.e.*, the SUT generates the same flow of system calls), LazyFS can reproduce bugs in a deterministic way, which is the case for the bugs considered in this paper (except for *Bug #20*).

However, both SUTs and test harnesses like Jepsen may run multiple threads or introduce other forms of nondeterminism—for instance, by generating a test workload of random transactions. Therefore, fault injection may not reproduce the same results for each run because LazyFS only addresses the nondeterminism of the OS's cache flushing mechanisms. Nevertheless, as shown in *Bug #20*, when combined with testing tools like Jepsen, LazyFS increases the probability of reproducing bugs. Finding and reproducing bugs in nondeterministic environments is a non-trivial research challenge that we defer to future work. Similarly, integrating LazyFS with concurrency debugging tools, useful for finding bugs in multi-threaded applications, would be an interesting future research path.

# 4 EVALUATION

Our experimental evaluation of widely used open-source systems has two main goals. First, we validate the benefits of using LazyFS for reproducing and understanding the root cause of known bugs, even when their reporting is ambiguous. For this, we use the bug reports discussed in §2. Further, we validate LazyFS's utility in finding new vulnerabilities, either by replicating known faults in other systems (or significantly different versions of the same system), or through integration with other fault injection frameworks, namely Jepsen. Our experiments answer the following questions:

- **RQ1:** Is LazyFS capable of reproducing bugs when the fault's type and steps that originated the error are known?
- **RQ2:** When the fault and reproduction steps are unknown, can LazyFS be useful for aiding in their discovery?
- **RQ3:** Can LazyFS expand users' knowledge about the impact of faults on tested systems (*i.e.*, data loss, corruption)?
- **RQ4:** Can LazyFS be used to uncover bugs in other systems (or different versions of a system)?
- **RQ5:** Can LazyFS help correct bugs and validate mechanisms intended to prevent them?

**Table 2: Summary of the experimental evaluation, including the tested systems and their versions, the type and number of injected faults, the total number of tested bugs, and the validated crash consistency mechanism. *L*, *LT* and *NT* denote *lost, linear torn* and *non-linear torn* writes, respectively. *C.C. Mechs.* means *crash consistency mechanisms.***

| System | Versions | Faults | | | #Bugs | #C.C. Mechs. |
|---|---|---|---|---|---|---|
| | | *L* | *LT* | *NT* | | |
| **PostgreSQL** | 12.11, 15.2 | 1 | | 1 | 1 | 1 |
| **Redis** | 7.0.4, 7.2.3 | 1 | 1 | 1 | 1 | 1 |
| **ZooKeeper** | 3.4.8, 3.7.1 | 1 | | 2 | 3 | |
| **etcd** | 2.3.0, 3.4.25, 3.5, 3.6 | 2 | 4 | 3 | 7 | |
| **LevelDB** | 1.12, 1.15, 1.23 | 3 | | 1 | 4 | |
| **PebblesDB** | 1.0 | 2 | | 1 | 3 | |
| **Lightning N.** | 0.15.1 | 1 | | | 1 | |
| **Total** | ***15*** | ***11*** | ***5*** | ***9*** | ***20*** | **2** |

## 4.1 Methodology

Next we describe the methodology used in our experiments.

**Environment.** Experiments run on servers equipped with a 4-core Intel Core i3-4170, 16 GiB of RAM, a 128 GiB SSD, running Ubuntu 20.04 LTS with Linux kernel 5.4.0 and the `ext4` file system.

**Systems.** Table 2 indicates the evaluated systems and corresponding versions. In addition to the six systems discussed in §2, we also include PebblesDB [35], a key-value store built on top of LevelDB. When reproducing known bugs, systems are configured according to the instructions provided by the corresponding report. For new bugs, the default system configurations are used.

The selected SUTs were chosen as they are representative data-centric systems and databases that leverage the OS cache.

**Workloads.** Systems are tested with simple workloads that consist of inserting, updating, and reading data. Further, the SUTs are always initialized to a known state (*i.e.*, empty or with a well-known set of values). No workload is applied in cases where the fault injection occurs at the systems' initialization phase.

**Fault Model.** Our experiments include the three type of faults described in §2, namely `lost` write(s), `linear torn` write(s) and `non-linear torn` write(s). Our model considers the injection of exactly one fault at each experiment.

Systems are configured to store their data under LazyFS's directory tree, while our tool is backed up by a regular `ext4` file system backend. To ensure a reproducible setup, we configure LazyFS's Page Cache to have enough space to hold all unflushed data. Also, the page and block size are configured to 4 KiB, a standard value in many file systems [4, 28, 49].

**Observations.** For each experiment, we observe how the system behaves by analyzing its post-fault state (*i.e.*, error/crash messages, data loss/corruption). This information is combined with the extra profiling output produced by LazyFS.

**Reproducibility.** After running each experiment (*i.e.*, setting up LazyFS and the SUT, running the workload, applying a given fault, and collecting metrics), we ensure that the setup is properly cleaned to avoid any impact in subsequent tests. In total, we conduct thirty-eight experiments, each including at least three repetitions. Experiments take on average 8.4 seconds, ranging from 1 second to 84 seconds. Notably, the small amount of time to run each experiment shows that LazyFS provides a time-efficient approach for fault injection. All steps and scripts to reproduce the findings detailed next are available in LazyFS's repository.

It is important to note that the methodology followed in our experiment combining Jepsen and LazyFS is slightly different. Given the exploratory nature of Jepsen, multiple tests are executed, each running for several minutes and involving the injection of multiple `lost` write faults, as further detailed in §5.3.

## 5 RESULTS AND TAKEAWAYS

Table 3 summarizes the twenty fault scenarios reproduced with LazyFS. The table divides these across three different categories: five reported bugs with clear reproduction steps, seven bugs containing insufficient or ambiguous information on how to reproduce them, and eight new issues found with our tool.

Next, we explain how bugs are reproduced and found with LazyFS (§5.1, §5.2 and §5.3). Then we discuss common strategies to tolerate and recover from `lost` and `torn` write faults (§5.4), including experiments that validate the crash recovery mechanisms of PostgreSQL and Redis.

### 5.1 Known bugs

We start by discussing our findings for bug reports that include reproduction steps. Our fault injection strategy follows as closely as possible the workloads and instructions provided by users. Note that with LazyFS we do not require any source code changes or manual intervention to mimic the faults, which are common steps suggested in most of these reports (§2).

*Bugs #3* and *#5* mention data loss and missing `fsync` calls in LevelDB, which indicates potential vulnerabilities involving `lost` write faults. We reproduce *Bug #3* by crashing LazyFS immediately before LevelDB renames the file *000002.dbtmp*. Forcefully stopping LazyFS means that unsynced data in the file system is lost. When rebooting LevelDB, we find that important metadata from the *MANIFEST* file is lost, making the Key-Value Store (KVS) unable to boot.

For *Bug #5*, we also inject a `lost` write fault, but only after the workload (a mix of synchronous and asynchronous `PUT` requests) finishes and LevelDB shutdowns gracefully. Later, the KVS fails to reboot, pointing out to corruption of metadata stored in the *CURRENT* file. Two unexpected and interesting findings result from this experiment. First, the error message is not accurate. By leveraging LazyFS's output, we find that the bug is actually caused by data loss in other files of the KVS, specifically in *000005.sst* and *000003.log* files. Second, the report of *Bug #5* mentions a silent error, which is not what we observe. In fact, the error outputted by LevelDB is identical to the one mentioned in *Bug #4*.

**Table 3: Summary of the known, ambiguous, and new bugs reproduced with LazyFS, including the type of faults injected, the impact of the fault in the system, the error observed, and whether the bug is still present in recent versions.**

| | System | Tag | Fault | Impact | Error | Recent V. |
|---|---|---|---|---|---|---|
| Known | PostgreSQL | #1 | L | C | Corrupted File | ✓ |
| | LevelDB | #3 | L | U | Metadata Corruption | ✓ |
| | | #4,#5 | L | U | Metadata Corruption | ✓ |
| | | #6 | NT | C | Checksum Mismatch | × |
| Ambiguous | ZooKeeper | #2 | NT | U | Magic Number Mismatch | ✓ |
| | | #7 | L | U | Unexpected Exception | ✓ |
| | | #8 | NT | U | CRC Mismatch | ✓ |
| | etcd | #9,#10 | NT, LT | U | CRC Mismatch | × |
| | Redis | #11 | L | S | × | × |
| | Lightning N. | #12 | L | C | File Reading Error | ✓ |
| New | PebblesDB | #13 | L | U | Metadata Corruption | ✓ |
| | | #14 | L | S | × | ✓ |
| | | #15 | NT | C | Checksum Mismatch | ✓ |
| | etcd | #16 | LT | U | Invalid Database | ✓ |
| | | #17 | NT | U | Bus Error | ✓ |
| | | #18 | LT | U | Invalid File Size | ✓ |
| | | #19 | L | U | Missing File | × |
| | | #20* | L | I | × | ✓ |

**Properties**
– Not Applicable
✓ Observed
× Not observed
∗ Found with Jepsen

**Fault Type**
***L*** - Lost write
***LT*** - Linear Torn write
***NT*** - Non-linear Torn write

**Impact**
***U*** - Unavailability
***S*** - Silent Data Loss
***I*** - Data Inconsistency
***C*** - Data Corruption/Loss

A different approach is followed for *Bugs #1* and *#6*, in which we first conduct a *profiling run* with the SUT running on top of LazyFS without injecting faults. The goal is to observe unsynced data and the flow of operations issued by PostgreSQL and LevelDB to better understand the timing and placement of faults. The workloads used during profiling are identical to those used for fault injection.

For PostgreSQL (*Bug #1*), we find that the statistics used to optimize query plans are never explicitly flushed to the *pgstat.stat* file. Although the database's documentation states that a permanent copy of the statistics is stored when the server shuts down cleanly [42], we show that this does not hold true when a `lost` write fault is injected by LazyFS.

As for LevelDB (*Bug #6*), when inserting a large key-value pair (*i.e.*, ≈44 KiB) as indicated in the report, we identify a sequence of five consecutive `write` calls to the commit log file that are vulnerable to `torn` write faults. By persisting only the first, third, and fifth calls and crashing LazyFS before the remaining ones are flushed, LevelDB detects data corruption through its checksum validation mechanism. While the KVS provides a *RepairDB* method to amend corrupted key-value pairs, we find that this mechanism simply merges the partially written non-linear information, resulting in an erroneous entry at the KVS.

### 5.2 Ambiguous bugs

Most of the studied bug reports do not include any reproduction steps and provide little to no information about errors reported

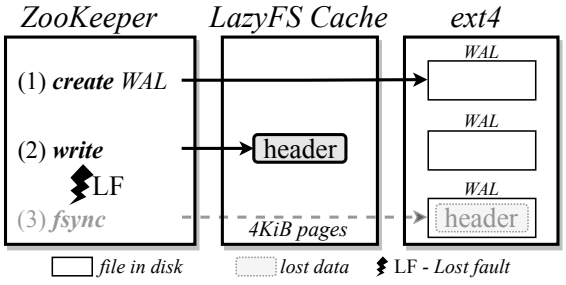


**Figure 5: Lost fault in ZooKeeper's *Bug #7*.**

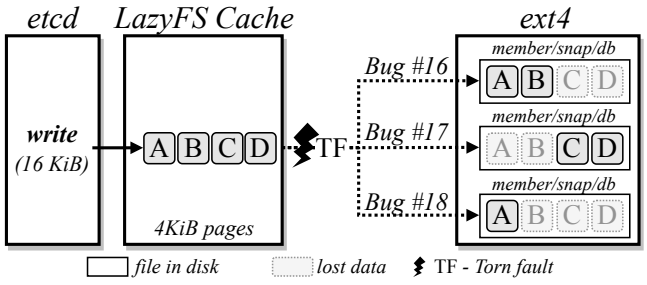


**Figure 6: Torn write faults in etcd's *Bugs #16, #17, #18*.**

by faulty systems. We leverage the information contained in the reports to build representative workloads and execute a *profiling run* to analyze potential scenarios for fault injection.

For *Bug #11*, we add a new user to the Access Control List (*ACL*) in Redis, while for *Bug #12*, we create a wallet in the Lightning Network system. Through profiling, we identify that both systems have cached data that is never synced explicitly to disk. We use LazyFS to inject a `lost` write fault after a graceful shutdown of these systems. After restarting Redis, the user no longer exist in the system because data at the *ACL* file is silently lost. In Lightning Network, users are unable to access the new wallet, and the system reports read errors for several files.

For *Bug #7*, the problem results from a crash during Zookeeper's bootstrap. As exemplified in Figure 5, we reproduce the bug by injecting a `lost` write fault and crashing LazyFS after data is written to the transaction log file but before the corresponding `fsync` call. Zookeeper is then unable to restart as a log file exists, but critical metadata from it is missing.

Reports for *Bugs #8*, *#9*, and *#10* point to `torn` writes as the root cause of the anomalies. Based on this hint, we use workloads that insert key-value pairs larger than a typical page size (*i.e.*, 4 KiB), and profile etcd and ZooKeeper.

We find that both systems perform sequences of write operations followed by an `fsync` call in their logs. Injecting `non-linear torn` write faults results in partially persisted data and makes Zookeeper and etcd unable to restart due to checksum mismatches at their respective transaction logs. We observe that the same error happens for large write operations issued by etcd when these are linearly and non-linearly torn. In ZooKeeper's case, applying its recovery script to fix this specific error proves to be ineffective.

Finally, by reproducing *Bug #2*, we find that Zookeeper is vulnerable to `non-linear torn` write faults when clients connect to the server component. At this stage, consecutive `write` calls are made to the server's transaction log file. Leaving this file with partially written content leads to Zookeeper being unable to restart.

**Testing recent versions.** We replicate (*i.e.*, use the same workloads and fault configurations) the aforementioned known and ambiguous bugs on more recent and still maintained versions of the corresponding systems. Eight of these bugs are still present in newer versions, while only *Bugs #6*, *#9*, *#10*, and *#11* are fixed.

Interestingly, *Bug #5* still suffers from the same fault but now results in silent data loss: LevelDB reports no error message. *Bug #7* now reports a missing snapshot file in Zookeeper. This is a misleading error message as the snapshot file exists in the file system; through LazyFS's profiling output, we know that the root cause is actually tied to data loss in the transaction log file.

## 5.3 New bugs

Next, we show how LazyFS is used to uncover new bugs. We consider as new the issues found with LazyFS that were previously unidentified. Note that old bugs still present in the systems' recent versions are not included in this classification.

Three strategies are employed in these experiments. First, we use LazyFS to replicate some of the previous fault injection setups in other (similar) systems. Then, we test systems' versions that change considerably the flow of I/O operations and become vulnerable to `lost` and `torn` write faults. Finally, we combine LazyFS with other bug exploration tools, specifically Jepsen.

**Testing other systems.** First, we test if bugs from LevelDB are present in PebblesDB [35], a KVS built on top of the former.

By injecting `lost` write faults, *i.e.*, before the `rename` call to file *000002.dbtmp* (*Bug #3*), and after the workload finishes and the KVS cleanly shutdowns (*Bug #5*), we observe two new bugs in PebblesDB (*Bugs #13* and *#14*). As expected, these bugs exhibit similar root causes and error behaviors as in LevelDB. *Bug #13* leads to metadata corruption and leaves PebblesDB unable to restart, while *Bug #14* causes silent data loss of key-value pairs.

Notably, PebblesDB changes some of the I/O patterns observed in LevelDB. For instance, it does not perform sequences of multiple `write` calls to the transaction log file (as in *Bug #6*). Instead, it issues a `write` call sizing 12 KiB. Injecting a `non-linear torn` write fault leads to a new data corruption bug in PebblesDB (*Bug #15*) [34].

Another approach to replicate bugs on similar systems is based on *Finding 5* from §2.2.2. Concretely, we test etcd's *WAL* file based on bugs from other systems that affect transaction logs (*Bug #7* from Zookeeper). A `lost` write fault after the first `write` call to the *WAL* triggers a new bug in etcd (*Bug #19*), where it fails to restart and reports a *"snapshot not found"* error. Once again, this is a misleading error as the issue's root cause is, in fact, linked to data loss at the *WAL* file.

**Testing new I/O flows.** Recent versions of etcd substantially change the flow of I/O operations. Notably, we find a large write operation of 16 KiB made at the KVS bootstrap to the file *member/snap/db*. As depicted in Figure 6, by injecting different combinations of `torn` write faults, we discover three new bugs.

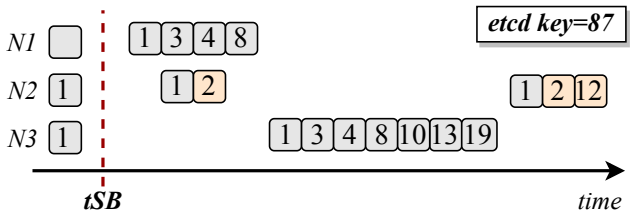


**Figure 7: Split brain scenario in ectd's *Bug #20*.**

When the large write is divided into two parts of 8 KiB (each part spreading across two 4 KiB cache pages) and only the first part is persisted, etcd is unable to restart and exhibits an *"invalid database"* error (*Bug #16*). If only the second part is persisted, etcd fails upon restart with a *"BUS error"* (*Bug #17*).

We reported these two bugs to the etcd team [10]. The maintainers suggested fixing these by breaking down the large write into four independent operations (*i.e.*, aligned with the file systems' page size) and performing an `fsync` call after each write. Using LazyFS, we prove that the solution falls short of resolving the issue. Attempting to restart etcd after dividing the large write into four parts and persisting only the first results in a new *"file size too small"* error (*Bug #18*), which suggests that this large write needs to be persisted atomically [9].

**Exploring bugs with LazyFS and Jepsen.** To assess the usefulness of combining our solution with other injection tools, we integrated LazyFS with Jepsen and tested several systems [18–20].

Jepsen is a library that generates random operations (*e.g.*, reads and writes) to target a specific system. While executing these operations, it injects faults such as killing nodes, altering the network traffic, and changing clocks, among others. It also checks the operations' history to verify if certain invariants are satisfied—for instance, if all acknowledged writes are preserved.

With LazyFS, Jepsen can simulate power failures, which are very different from a process crash. If a node disconnects from the cluster after committing a specific transaction, there is a possibility that the transaction still resides in the node's page cache and be later flushed by the OS. However, in the event of a power failure, the node could lose that transaction, leading to inconsistencies in the cluster when the node reconnects.

We find *Bug #20* with Jepsen using LazyFS's ability to forget unsynced data. Our workload involves randomly-generated transactions that read or appended elements to specific keys. During execution, `lost` write faults are injected every 15 seconds. The fault injection process involves Jepsen forcefully stopping the SUT and sending a `clear-cache` command to LazyFS.

As shown in Figure 7, these simulated crashes cause data inconsistency within etcd's cluster where replicas start to diverge in the value for a given key, suggesting a *split-brain* scenario. More specifically, after a given time (*tSB*), one of the replicas (*N2*) has for the same key (*87*) the values *[1,2,...]*, while the others have *[1, 3, 4, 8, ...]*. The etcd team confirmed this bug in version 3.4, but has not yet identified a specific cause for it [8].

**Confirmed bugs.** With LazyFS, the development teams reproduced and confirmed four of the new bugs: #16, #17, #18 and #20 [8–10].

## 5.4 Validate crash consistency mechanisms

Developers employ different strategies to protect and recover from the `lost` and `torn` write faults studied in this paper. These strategies depend on the data durability requirements of each system.

**Common bug fixes.** When the goal is to ensure the persistence of all written data at a certain stage of the system's execution, `fsync` must be called for the respective files (as done to fix *Bug #11*).

However, as shown in the paper, this strategy is not always sufficient. When a sequence of `write` calls or large writes exceeding the file systems' page size must be persisted in a time-orderly manner, developers may need to ensure that each write operation fits on a single file system's page and is flushed to disk explicitly before calling the next operation. While this strategy is vulnerable to `linear torn` writes (*i.e.*, under crashes, data may be partially persisted but in an ordered fashion), it may be sufficient for some systems to function properly and recover from this faulty scenario (*Bug #6*).

A more complex approach is required when systems need *atomicity* across a sequence of operations (*e.g*, multiple writes, file creation plus writing), or for large writes. A common strategy involves writing data to a temporary file (*e.g.*, *log.tmp*), explicitly syncing its data and then renaming it (`rename` is an atomic operation in most POSIX file systems) to the final file (*e.g.*, *log*). This approach guarantees that when updated, *log* will include all the new written content or none of it. It is used by etcd (*Bug #19*) to prevent an empty *WAL* file under crashes (*i.e.*, the file is only created if the first `write` is successfully persisted). Atomic update patterns are also suggested for fixing *Bugs #7* and *#18*.

**Protection and recovery.** PostgreSQL and Redis use different mechanisms to protect and recover from `torn` write faults. Using LazyFS, we conducted experiments to assess their correctness.

PostgreSQL stores information from tables and indexes in disk pages, each with 8 KiB. To ensure their *atomicity*, *i.e.*, that these do not end up partially written and the database becomes corrupted, it provides a configuration where pages and updates to these are ensured to be first written to a *WAL* file, along with their checksums. When a crash occurs, partially written disk pages can be fixed with the ones written in the *WAL* file, while partially written *WAL* pages are discarded due to checksum mismatch [41].

We use LazyFS to inject a `non-linear torn` write fault in PostgreSQL's write operations to its disk pages while the previous configuration is disabled and enabled. In both cases, the database is able to restart. However, running the PostgreSQL *amcheck* module [40] with the configuration disabled, to verify the logical coherence of relation structures, results in a database corruption error.

Redis provides two different crash protection mechanisms: the periodic creation of dataset snapshots, and an *Append Only File (AOF)* that logs every write operation received by the server. Using a simple workload to insert and update a key-value pair with 10 KiB, we observe both sequences of multiple `write` calls and large writes being issued to the *AOF* file. When subjected to `linear torn` write faults, Redis successfully restarts after detecting partially written values in the file and automatically discarding them. However, for `non-linear torn` write faults, Redis crashes but provides a tool for users to identify and discard corrupted values.

## 5.5 Discussion

Our experimental evaluation proves LazyFS' usefulness for testing systems under total and partial data loss.

**Reproduction and discovery.** LazyFS is useful for reproducing known and ambiguous bugs and uncovering new issues. First, for the bugs with clear reproduction instructions, LazyFS eliminates error-prone and complex steps such as unplugging power cables, modifying the system's source code, and deploying complex setups (RQ1). Second, LazyFS's output helps to identify possible points of vulnerability in systems, which allows the reproduction of bugs when users are not sure about the steps that led to them (RQ2). Finally, we discover new issues with LazyFS by following different approaches that developers can also adopt to validate if their systems are crash-consistent (RQ4).

**Validation.** LazyFS also helps validate the correctness of solutions for protecting and recovering against data durability issues (RQ5). As shown in §5.3, we test etcd team's suggestion to fix *Bugs #16* and *#17* and confirm that it does not solve the problem. Additionally, LazyFS can be used to test crash consistency mechanisms, which is essential to prevent broken mechanisms that are ineffective (*Bug #8*) or end up worsening the nefarious effects of errors (*Bug #6*).

**Impact and error report.** `Lost` and `torn` write faults have different impacts: unavailability, data loss, corruption, and inconsistency. While total data loss can go unnoticed (*e.g.*, *Bug #11*), partial data loss usually causes systems' errors (*e.g.*, *Bug #2*). However, some systems often report misleading errors, which point to a possible root cause when the problem is actually elsewhere (*Bugs #5*, *#7*, *#19*). LazyFS aids in this matter, providing additional information that helps unveil the concrete bug causes (RQ3).

# 6 RELATED WORK

Works related to LazyFS address data durability and crash consistency guarantees at three different levels, *i.e.*, the storage device, the file system, and the applications and systems using these.

**Storage devices.** Zheng et al. [57] assesses the reliability of SSDs and HDDs under faults, while Tseng et al. [51] examines post-fault data integrity in flash memory. Both employ hardware-based fault injection to find bugs (*e.g.*, data corruption) at the device level.

**File systems.** Other solutions focus on validating the correctness of file systems' internals and APIs. Ferrite [2] uses formal models, while FiSC [55] and EXPLODE [54] use model checking for testing how different properties of file systems (*e.g.*, I/O *ordering*, *atomicity*) hold under faulty scenarios. EXPLODE is not limited to file systems, as its checker can be used to validate different storage stacks (*e.g.*, RAID systems, file systems, databases). $B^3$ [29] follows a different approach consisting of the exhaustive generation of fault injection workloads (*i.e.*, including crashes after persistence operations such as `fsync`) to find crash consistency bugs.

LazyFS differs from and complements these works by focusing on the impact of faults at the application/system level (*e.g.*, databases, key-value stores, blockchains), even when file systems and devices are considered to be bug-free.

**Applications/Systems.** The works that most resemble LazyFS, goal-wise, aim at exploring data durability issues caused by erroneous interactions of applications and systems with the storage stack (*e.g.*, file system). For instance, EXPLODE's model checker can also be used for identifying erroneous data *durability* assumptions (*e.g.*, *atomicity* of `write` system calls) made by systems.

Zheng et al. [56] and ALICE [36] take a different approach. Both use the record and replay strategy, which involves running a set of representative workloads on top of the SUT and then replaying the collected traces and exploring different fault scenarios to uncover violations of constraints (*e.g.,* ACID properties of transactions in Zheng et al., `lost` and `torn` writes in ALICE). ALICE also resorts to model checking for specifying the properties (*i.e.*, requests *ordering* and *atomicity*) of different file system implementations.

As explained in §2, these solutions are complementary to LazyFS. Comprehensively exploring the behavior of systems to uncover data durability bugs is of extreme importance (*e.g.*, during the design and implementation phases), but it may not be a time-efficient approach when developers wish to reproduce bugs reported by users, which is our tool's main goal.

CuttleFS [44] and ErrFS [13] provide simple and efficient fault injection mechanisms. As in LazyFS, CuttleFS implements an internal page cache, but the goal is to mimic failed `fsync` calls, at the file system level, and observe how the above applications and systems handle these. ErrFS mimics typical I/O errors reported by read and `write` system calls. It is used to test the capacity of distributed systems to recover from single-node faults.

LazyFS differs from and complements these solutions because it injects different types of faults: `lost` and `torn` writes. Further, we show that while targeted for single-node fault injection, our tool can be integrated and complement distributed testing frameworks, such as Jepsen [16].

# 7 CONCLUSION

This paper presents LazyFS, a new fault injection tool that enables users to test the resiliency of their systems against `lost` and `torn` write faults. We study twelve reported bugs from widely used systems and show that our tool is key to understanding the root cause of ambiguous errors while automating and facilitating their reproduction. Further, we show that LazyFS can be used independently, or in combination with the Jepsen framework, to find eight new bugs in the PebblesDB and etcd systems.

LazyFS, along with all the documentation to reproduce the bugs discussed in the paper, is available at https://github.com/dsrhaslab/lazyfs. Our tool has been incorporated by the ectd and MongoDB teams into their testing and debugging environments. We encourage other developers and researchers to use LazyFS to validate the data durability of their solutions.

# ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020.

# REFERENCES

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[2] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st International Conference on Architectural Support*

*for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/2872362.2872406

[3] Daniel Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3 ed.). Oreilly & Associates Inc.

[4] BTRFS. 2024. Status. https://btrfs.readthedocs.io/en/latest/Status.html Last accessed on July 18, 2024.

[5] Gyeongyeol Choi and Youjip Won. 2018. Analysis for the Performance Degradation of Fsync() in F2FS. In *Proceedings of the 9th International Conference on E-Education, E-Business, E-Management and E-Learning*. Association for Computing Machinery, New York, NY, USA, 71–75. https://doi.org/10.1145/3183586.3183605

[6] etcd. 2016. ETCD data gets corrupted with error "read wal error (walpb: crc mismatch)". https://github.com/etcd-io/etcd/issues/6191 Last accessed on July 18, 2024.

[7] etcd. 2019. ETCD data gets corrupted with error "walpb: crc mismatch". https://github.com/etcd-io/etcd/issues/11488 Last accessed on July 18, 2024.

[8] etcd. 2022. Possible split-brain & loss of committed write with loss of un-fsynced data. https://github.com/etcd-io/etcd/issues/14143 Last accessed on July 18, 2024.

[9] etcd. 2023. Avoid torn init writes. https://github.com/etcd-io/bbolt/issues/567#issuecomment-1792646307 Last accessed on July 18, 2024.

[10] etcd. 2023. etcd fails to start after power failure. https://github.com/etcd-io/etcd/issues/16596 Last accessed on July 18, 2024.

[11] etcd. 2024. Add support for lazyfs. https://github.com/etcd-io/etcd/pull/14691 Last accessed on July 18, 2024.

[12] etcd. 2024. Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd Last accessed on July 18, 2024.

[13] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies*. Association for Computing Machinery, New York, NY, USA, 149–166. https://doi.org/10.1145/3125497

[14] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *Comput. Surveys* 13, 2 (1981), 223–242. https://doi.org/10.1145/356842.356847

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[16] Jepsen. 2024. A framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen Last accessed on July 18, 2024.

[17] Jepsen. 2024. Analyses. https://jepsen.io/analyses Last accessed on July 18, 2024.

[18] Jepsen. 2024. Jepsen Report: MySQL 8.0.34. https://jepsen.io/analyses/mysql-8.0.34 Last accessed on July 18, 2024.

[19] Jepsen. 2024. MongoDB Jepsen tests. https://github.com/jepsen-io/mongodb Last accessed on July 18, 2024.

[20] Jepsen. 2024. Tests for Percona Server with Group Replication Resources. https://github.com/jepsen-io/percona-gr Last accessed on July 18, 2024.

[21] LevelDB. 2014. "LevelDBError: Corruption: CURRENT file does not end with newline" database recovery was not possible. https://github.com/google/leveldb/issues/75 Last accessed on July 18, 2024.

[22] LevelDB. 2014. Possible bug: After a power-loss, LevelDB does not discard partially-flushed transactions. https://github.com/google/leveldb/issues/251 Last accessed on July 18, 2024.

[23] LevelDB. 2014. Possible bug: Missing a fsync() on the ".log" file before compaction. https://github.com/google/leveldb/issues/193 Last accessed on July 18, 2024.

[24] LevelDB. 2014. Possible bug: Missing a fsync() or msync() call after creating MANIFEST-000001. https://github.com/google/leveldb/issues/189 Last accessed on July 18, 2024.

[25] LevelDB. 2024. A fast key-value storage library. https://github.com/google/leveldb Last accessed on July 18, 2024.

[26] libfuse. 2024. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse Last accessed on July 18, 2024.

[27] Lightning Network Daemon. 2024. A complete implementation of a Lightning Network node. https://github.com/lightningnetwork/lnd Last accessed on July 18, 2024.

[28] Linux man page. 2024. mkfs.xfs(8) - Linux man page. https://linux.die.net/man/8/mkfs.xfs Last accessed on July 18, 2024.

[29] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 33–50.

[30] MongoDB. 2024. Extend WT fault injection testing to support LazyFS. https://jira.mongodb.org/browse/WT-9368 Last accessed on July 18, 2024.

[31] Lightning Network. 2022. Replace ioutil.WriteFile usage with Fsync and friends. https://github.com/lightningnetwork/lnd/issues/6720 Last accessed on July 18, 2024.

[32] The Linux Kernel Organization. 2024. open(2) — Linux man page. https://man7.org/linux/man-pages/man2/open.2.html Last accessed on July 18, 2024.

[33] The Linux Kernel Organization. 2024. write(2) — Linux man page. https://man7.org/linux/man-pages/man2/write.2.html Last accessed on July 18, 2024.

[34] PebbledDB. 2023. PebblesDB does not discard partially-flushed values. https://github.com/utsaslab/pebblesdb/issues/28 Last accessed on July 18, 2024.

[35] PebblesDB. 2024. The PebblesDB write-optimized key-value store (SOSP 17). https://github.com/utsaslab/pebblesdb Last accessed on July 18, 2024.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[37] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *Queue* 13, 7 (2015), 20–28. https://doi.org/10.1145/2800695.2801719

[38] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. 2017. SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All. In *Proceedings of the 10th ACM International Systems and Storage Conference*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3078468.3078480

[39] PostgreSQL. 2012. "global/pgstat.stat" corrupt. https://www.postgresql.org/message-id/flat/17700.1340394540%40sss.pgh.pa.us#b9644ae1d7f27171bb6921d0a7084fa4 Last accessed on July 18, 2024.

[40] PostgreSQL. 2024. amcheck — tools to verify table and index consistency. https://www.postgresql.org/docs/current/amcheck.html Last accessed on July 18, 2024.

[41] PostgreSQL. 2024. Full page writes in PostgreSQL. https://www.postgresql.org/docs/15/runtime-config-wal.html#GUC-FULL-PAGE-WRITES Last accessed on July 18, 2024.

[42] PostgreSQL. 2024. The Statistics Collector. https://www.postgresql.org/docs/12/monitoring-stats.html Last accessed on July 18, 2024.

[43] PostgreSQL. 2024. The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/ Last accessed on July 18, 2024.

[44] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Can Applications Recover from fsync Failures? *ACM Transactions on Storage* 17, 2 (2021), 1–30. https://doi.org/10.1145/3450338

[45] Redis. 2022. Make cluster config file saving atomic and fsync acl. https://github.com/redis/redis/pull/10924 Last accessed on July 18, 2024.

[46] Redis. 2024. In-memory database that persists on disk. https://github.com/redis/redis Last accessed on July 18, 2024.

[47] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 1–16.

[48] strace. 2024. strace: linux syscall tracer. https://strace.io Last accessed on July 18, 2024.

[49] The Linux Kernel Archives. 2024. ext4 Data Structures and Algorithms. https://www.kernel.org/doc/html/latest/filesystems/ext4/overview.html#blocks Last accessed on July 18, 2024.

[50] The Linux Kernel Organization. 2024. fsync(2) - Linux man page. https://man7.org/linux/man-pages/man2/fsync.2.html Last accessed on July 18, 2024.

[51] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. 2011. Understanding the Impact of Power Loss on Flash Memory. In *Proceedings of the 48th Design Automation Conference*. Association for Computing Machinery, New York, NY, USA, 35–40. https://doi.org/10.1145/2024724.2024733

[52] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or not to FUSE: performance of user-space file systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 59–72.

[53] Ric Wheeler and Red Hat. 2009. How to (Not) Lose Your Data. In *Proceedings of the Linux Symposium*. 303–310. https://kernel.org/doc/ols/2009/ols2009-pages-303-310.pdf Last accessed on July 18, 2024.

[54] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) *(OSDI '06)*. USENIX Association, USA, 10.

[55] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Trans. Comput.*

*Syst.* 24, 4 (2006), 393–423. https://doi.org/10.1145/1189256.1189259

[56] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 449–464.

[57] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDS under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'13)*. USENIX Association, USA, 271–284.

[58] ZooKeeper. 2013. Transaction log: /logs/jhe/nc/apps/zookeeper/version-2/log.1400470027 has invalid magic number 0 != 1514884167. https://lists.apache.org/thread/dlhrd5t2xyjr425l78xofqs7r3v4s486 Last accessed on July 18, 2024.

[59] ZooKeeper. 2015. Zookeeper failed to start for empty txn log. https://issues.apache.org/jira/browse/ZOOKEEPER-2332 Last accessed on July 18, 2024.

[60] ZooKeeper. 2016. Possible Cluster Unavailability. https://issues.apache.org/jira/browse/ZOOKEEPER-2560 Last accessed on July 18, 2024.