

Patch and Bug Analysis of Linux Filesystems in the Context of Modern Storage Evolution

Jianjie Zheng
School of Informatics
Xiamen University
Xiamen, China

Mingyu Li
School of Informatics
Xiamen University
Xiamen, China

Chenyang Huang
School of Informatics
Xiamen University
Xiamen, China

Hongfei Ying
School of Informatics
Xiamen University
Xiamen, China

Shuangyang Han
School of Informatics
Xiamen University
Xiamen, China

Abstract—Over the past decade, the rapid evolution of the Linux storage stack has expanded file system responsibilities to maintaining consistent file abstractions across heterogeneous devices and complex protocol states. This architectural shift fundamentally alters how software defects are generated and mitigated. This paper presents a comprehensive, long-term empirical study of patch evolution and defect characteristics across five representative Linux file systems (ext4, F2FS, ZoneFS, EROFS, and NFS) using Git history from January 2016 to April 2026. We construct a cross-system aligned empirical framework that correlates patch types, bug categories, and five structural responsibility domains, leveraging SZZ-based latency analysis and code churn metrics to quantify latent technical debts. Based on empirical mining of 3,906 bug-fixing commits, our findings demonstrate that file system defect patterns are tightly coupled with their underlying hardware assumptions and concurrency models. Through an archeological deep dive into long-latency and high-cost patches, we classify three critical failure modes: cross-subsystem contract mismatches, feature matrix interweaving, and asynchronous horizon detachment. Finally, we distill five defensive programming guidelines, providing quantitative empirical foundations for test design, code review, and the sustainable maintenance of modern storage systems.

Index Terms—Linux filesystems, storage systems, software evolution, bug analysis

I. INTRODUCTION

The file system is located at the core of the operating system’s storage stack. Its implementation quality directly affects data reliability, I/O performance, and the fault isolation capabilities of upper-layer applications. Over the past decade, the hardware and deployment environments faced by the Linux file system have undergone significant changes: on one hand, the traditional block device abstraction still supports a large number of general scenarios; on the other hand, Flash/SSD, zoned block device, compressed read-only image, and networked storage have gradually become important load sources. This change has expanded the responsibilities of the file system from “implementing general file semantics on top of block devices” to “maintaining consistent, reliable, and high-performance file abstraction among heterogeneous devices, protocol states, and high-concurrency I/O paths”.

This evolution has changed the way file system defects are generated and repaired. Mature file systems typically have a long history of compatibility and complex metadata invariants; Flash-oriented file systems need to coordinate segment management, garbage collection and checkpoints; zoned device

oriented file systems must comply with sequential writing and zone state constraints; read-only compressed file systems need to maintain consistency in compression layout, page cache and read path; network file systems also need to handle RPC transmission, remote state and client cache semantics. The different architecture choices do not only bring about performance differences, but also reflect in patch distribution, bug types, bug latency and repair costs. Therefore, studying the patch and defect characteristics of Linux file systems during their long-term evolution process is helpful in understanding the engineering complexity and maintenance risks of modern storage systems.

Existing research has provided an important foundation for such problems. “A Study of Linux File System Evolution” from FAST ’13 analyzed 5,079 file system patches from the Linux 2.6 era, revealing the proportional relationships among bug fixes, maintenance, performance optimization, and reliability enhancement in file system development, and further pointed out that semantic errors and concurrency errors constitute significant components of file system defects. [1] This work demonstrates that patches accepted into the mainline kernel not only record code changes but also document the actual engineering costs paid by developers in terms of reliability and maintainability. However, the file systems covered by the FAST ’13 study mainly belong to the era of traditional local file systems and do not yet fully reflect the new problems brought about by hardware specialization, compressed read-only deployment, and networked access in the Linux storage stack over the past decade. As file systems such as F2FS, ZoneFS, and EROFS have entered the mainline and continued to evolve, it is necessary to re-examine the patch structure and defect patterns of modern Linux file systems.

This paper takes the Linux Git history from January 2016 to April 2026 as the research object and analyzes five file systems: ext4, F2FS, ZoneFS, EROFS, and NFS. They represent a mature general-purpose journaling file system, a Flash-friendly file system, a zoned block device file system, a read-only compressed file system, and a network file system client, respectively. This selection covers a variety of design points ranging from traditional block devices to modern heterogeneous storage and network protocols, enabling us to compare patch evolution and defect distribution under different architectural objectives.

Centered around the above objective, this paper focuses on five research questions.

RQ1: How are patch types distributed across different file systems? File system commits may be used to fix bugs, implement features, optimize performance, enhance reliability, adapt to hardware or protocols, or simply perform maintenance refactoring. The distribution of patch types can reflect the evolutionary stage and primary engineering pressures of a file system.

RQ2: Why do bug types differ across different file systems? This paper classifies bugs into semantic errors, concurrency errors, memory errors, and hardware interaction errors. The distribution of bug types across different file systems can reflect their state machine complexity, concurrency path complexity, object lifecycle management difficulty, and the degree of tightness in interacting with devices or protocols.

RQ3: What is the relationship between bugs and code responsibilities? A file system consists of multiple responsibility modules, such as metadata management, data I/O, space allocation, consistency recovery, cache management, compression, protocol interaction, and background tasks. Counting bugs by file system can only reflect overall differences, whereas counting them by code responsibility allows for further pinpointing the areas where defects are concentrated.

RQ4: Do bug latent periods differ across different types of code? The time interval between bug introduction and fix reflects the difficulty of defect exposure. Long-latent bugs often reside in exception paths, hardware-dependent paths, or concurrency paths that are difficult to cover, and they carry stronger implications for testing and code review.

RQ5: How do the fix costs differ across different bug types? Fix cost can be characterized by the number of modified files, the number of lines added and deleted, and code churn. High-cost bugs typically involve cross-module state adjustments, core invariant modifications, or complex error path fixes. Manual review of these commits can help summarize high-risk programming patterns in file system development.

Based on the current data mining, this paper has identified 3,906 bug fix commit–file system pairs from January 2016 to April 2026. Preliminary statistics show significant differences in the bug structures of different file systems: F2FS has the highest number of bug fixes, among which semantic errors account for the largest proportion; ext4 has a relatively high proportion of memory errors; hardware interaction errors account for more than half in ZoneFS; defects in EROFS are concentrated in hardware interaction and memory management; NFS exhibits a relatively balanced distribution across the four bug categories of semantics, concurrency, memory, and hardware interaction. These results indicate that the defect patterns of file systems are closely related to their hardware assumptions, data layouts, concurrency models, and protocol boundaries.

The main contributions of this paper are as follows.

First, this paper constructs a patch evolution analysis framework for modern Linux file systems. The framework not

only counts bug fixes but also classifies target commits into types such as bug fixes, feature development, performance optimization, reliability enhancement, hardware/protocol adaptation, and maintenance refactoring, thereby characterizing the evolutionary state of file systems from the overall patch distribution.

Second, this paper proposes a defect analysis method that combines bug types with code responsibilities. On top of classifying bugs into four categories—semantic, concurrency, memory, and hardware interaction errors—this paper introduces a unified code responsibility model to compare the distributional differences of defects across functional modules in different file systems.

Third, this paper incorporates bug latent periods and fix costs into file system maintenance risk analysis. This paper uses the "Fixes:" tag and the SZZ approach to estimate bug introduction time [2] and employs patch churn to measure fix cost, thereby identifying defects with long latency and high repair costs, providing a basis for test design and code review.

II. METHODOLOGY

This section introduces the data source, research objects, patch classification, bug classification, code responsibility attribution, bug latent period calculation, and fix cost measurement. The overall process consists of four stages: first, extracting target file system commits from the Linux Git history; second, classifying the commits by patch type; third, annotating bug fix commits with defect types and code responsibilities; and finally, calculating bug latent periods and fix costs, and generating statistical tables and charts.

A. Data Source and Research Objects

The research object is the Linux kernel Git repository. The time scope is limited to January 1, 2016 to April 30, 2026. This time window covers the main evolutionary process of Linux file systems over the past decade, and also covers the sustained development phases of Flash-friendly file systems, zoned block device support, read-only compressed deployment, and network file system clients.

This paper analyzes five file system directories: fs/ext4, fs/f2fs, fs/zonefs, fs/erofs, and fs/nfs. Among them, ext4 serves as a mature general-purpose journaling file system and is used to provide a traditional file system baseline; F2FS represents Flash/SSD-oriented specialized design; ZoneFS represents file systems that directly expose hardware physical constraints; EROFS represents read-only compression and image deployment scenarios; and NFS represents a network file system client. For NFS, this paper analyzes only fs/nfs/, excluding fs/nfsd/ and fs/nfs_common/, to avoid unclear attribution caused by mixing client-side, server-side, and common code.

Merge commits are excluded during commit extraction. Merge commits often aggregate multiple independent modifications, making their commit messages and modification scales difficult to correspond to a single engineering purpose, and are thus unsuitable for patch type and fix cost analysis. For each non-merge commit, the script records fields including

commit hash, author time, committer time, commit title, full commit message, modified file paths, lines added, lines deleted, and number of modified files. If a commit modifies multiple target file systems simultaneously, it is expanded and recorded as "commit-file system pairs" to ensure that modifications in different subsystems can be counted separately.

B. Patch Extraction and Preprocessing

Regarding the definition of Patch, in computer science and software engineering, a patch typically refers to a small software update used to fix bugs or vulnerabilities in a program. In this study, Patch is defined as the change content of a single file contained in one code commit. That is, if a commit modifies multiple files, it corresponds to multiple Patches. The reason for adopting this definition, rather than treating the entire commit as one Patch, is that our study of file system evolution focuses on specific modifications at the code level. Within the same commit, modifications to different files may serve the same function or fix, but at the code level, the change logic, impact scope, and modification type of each file may be completely different. Refining the granularity to the file level allows for more precise analysis of localized code changes, such as which files are frequently modified or which specific functions in a file are frequently fixed. At the same time, this also facilitates subsequent mapping of each Patch to specific developers, module ownership, and statistics of lines added and deleted. The Patch definition approach in this study ensures, in data statistics, that commits with more extensive changes exhibit more extensive impact, thereby supporting finer-grained data mining and trend analysis and facilitating the identification of code evolution trend characteristics.

The data acquisition in this study is based on the official Linux kernel Git repository. Benefiting from the relatively small size of the Linux system, this study used Git commands to clone the complete Linux kernel repository locally. After cloning, the local repository contains all commit records and the complete code history since the inception of the Linux kernel.

After obtaining the local repository, the corresponding source code directory paths were set for the five target file systems, namely F2FS, EXT4, ZONEFS, EROFS, and NFS. The path for F2FS is `fs/f2fs`, for EXT4 is `fs/ext4`, for ZONEFS is `fs/zonefs`, for EROFS is `fs/erofs`, and for NFS is `fs/nfs`. For each file system, combined with path filtering parameters, the list of hash values of all commits that modified any file under that directory was directly obtained. This step leverages Git's native path filtering capability, avoiding the traversal of all commits in the entire repository and greatly improving data acquisition efficiency.

After obtaining the list of commit hash values, Git commands were used again to retrieve the complete information for each commit, including the author name, author email, author date, committer name, committer email, committer date, and the complete commit message. At the same time, the statistics of lines added and deleted and the file path for each file modified by the commit were obtained. Subsequently,

for each modified file, the complete diff content of that file in this commit was obtained. The diff content of each file was individually extracted and stored.

At the data storage level, this study defines two granularities. The Commit level records hash value of each commit, author information, date, complete commit message, total number of modified files, total lines added and deleted, and the list of involved file paths. The Patch level records detailed information for each modified file, including the hash value of the owning commit, file path, modification type, lines added and deleted, and the complete diff content. The two levels are associated through the commit hash value. Prior to classification processing, all extracted raw data are stored in CSV format.

C. Patch Classification

In this study, based on the analysis of a large number of patches and community understanding, patches are classified into three main types: Security, Feature, and Maintenance. The Security type specifically refers to fixes or hardening related to system security, stability, or data integrity. This includes, but is not limited to, fixing known security vulnerabilities (e.g., CVE), fixing memory safety errors (e.g., use-after-free, double-free, buffer overflow), fixing privilege bypass, data leakage, exploitable crashes caused by null pointer dereference, and fixing security issues arising from concurrency race conditions. The core characteristic of the Security type is that if the patch is not applied, the system faces the risk of malicious exploitation or data corruption.

The Feature type refers to adding new capabilities, characteristics, or support to the file system. This includes adding new system call interfaces, new mount options, new IOCTL commands, supporting new encryption mechanisms, supporting new compression algorithms, adding adaptation to new hardware or new storage media, and implementing file system operations that were previously unsupported. The core characteristic of the Feature type is extending the behavioral boundary of the system on the existing basis and providing capabilities that were not available before.

The Maintenance type covers all other modifications except Security and Feature. Specifically, it includes: ordinary bug fixes (crashes or logic errors that do not lead to security vulnerabilities), code refactoring (renaming variables, splitting functions, adjusting code structure without changing behavior), performance optimization (reducing memory usage, lowering CPU overhead, optimizing I/O paths), code cleanup (deleting unused functions, fixing spelling errors, adjusting indentation and comments), compilation fixes (adapting to different kernel versions or compilers), and additions or deletions of test code. The core characteristic of the Maintenance type is the improvement of code quality or maintainability without directly introducing new features or fixing security vulnerabilities.

The reason for choosing these three types is that Security, Feature, and Maintenance are the three most core dimensions in the code evolution process. Security represents the reliability baseline of the system, Feature represents the capability

evolution of the system, and Maintenance reflects the health and maintainability of the code. The boundaries among the three are relatively clear and not prone to ambiguity, while also being able to well summarize the intent of the vast majority of commits in kernel development. In a complex project like the Linux kernel, many commits contain multiple types of changes simultaneously, such as fixing an unrelated small bug while adding a new feature, but we adhere to the kernel development norm that each patch should do only one thing, and therefore most commits can be clearly classified into one of the above three categories.

Regarding the classification method for Commits and Patches, this study adopts a manual classification method assisted by LLM judgment. Specifically, we wrote specialized classification prompts, requiring the model to determine the type of a commit based on the descriptive content in the commit message. The classification basis mainly consists of two points: first, whether the commit message contains security-related keywords, such as CVE, use-after-free, leak, overflow, vulnerability, bypass, race, etc.—if so, it is classified as Security; second, whether it introduces new features, such as add support, new feature, implement, enable, etc.—if so, it is classified as Feature; the remaining cases are classified as Maintenance. For cases where the commit message information is incomplete or the expression is ambiguous, the model makes inferences based on contextual semantics. In this study, the classification results of commits are typically generalized to the patches they cover, and this study obtains the fuzziness coefficient of the model when making commit classification judgments, enabling the assessment of the likelihood that the content covered by a commit exists across different category dimensions. At the same time, a rule-based method is also adopted to judge reliability. We use this to analyze the specific content of some commits with unclear classification boundaries, thereby making more precise classification judgments at the granularity of individual patch files using manual methods. Through manual sampling verification, the classification accuracy is within an acceptable range.

D. Bug Identification and Classification

Bug fix commits are the core objects of defect analysis in this paper. A commit is identified as a bug fix if it meets one of the following conditions: the commit subject or body contains defect-related expressions such as fix, bug, regression, incorrect, wrong, invalid, fail, failure, error, crash, oops, panic, hang, corruption, deadlock, race, leak, overflow, underflow, use-after-free, null pointer, out-of-bounds, CVE, or Fixes:. If a commit also contains explicitly maintenance-oriented expressions such as cleanup, typo, spelling, whitespace, formatting, documentation, rename, refactor, or no functional change, it is excluded by default; however, when a commit contains strong defect evidence such as Fixes:, crash, panic, race, deadlock, or use-after-free, it is retained.

For the identified bug fix commits, this paper further classifies them into four categories.

Semantic errors refer to defects where the file system logic is inconsistent with the expected semantics, commonly found in logic such as state judgment, return value handling, error code propagation, metadata consistency, inode/dentry state, mount/recovery, quota, journal, and orphan list. Concurrency errors refer to synchronization and timing issues among multiple execution contexts, including race conditions, deadlocks, lock ordering errors, atomicity violations, RCU usage errors, and workqueue/waitqueue timing errors. Memory errors refer to problems with object lifecycle, pointer safety, and boundary access, including null pointer dereference, memory leak, use-after-free, double free, uninitialized memory, out-of-bounds access, reference counting errors, and page/folio lifecycle errors. Hardware interaction errors refer to problems caused by incorrect interaction between the file system and underlying devices, I/O paths, or external protocol endpoints; for local file systems, this category includes issues with bio, block device, sector, zone, discard, flush, writeback, direct I/O, DAX, page cache, and readahead; for NFS, this category also includes network protocol interaction issues such as RPC, XDR, socket, transport, timeout, RDMA, and server state.

This paper adopts a single-label primary category strategy, where each bug fix commit is assigned to only one primary type. The priority order is: Memory, Concurrency, Hardware-Interaction, Semantic

Memory and concurrency errors typically have more explicit diagnostic terminology and failure modes; hardware interaction errors reflect device or protocol constraints; semantic errors have the broadest scope and serve as the default category when a commit cannot be classified into the first three. Adopting a single-label strategy ensures clarity in proportion statistics, but it also implies that some cross-category defects will be simplified to the dominant cause. This limitation will be discussed in the threats to validity.

E. Code Responsibility Attribution

To enable horizontal comparison of defect characteristics across heterogeneous file systems, this paper constructs a code attribution model based on responsibility alignment (Architectural Responsibility Attribution). Since the code organization forms and design philosophies of systems such as ext4, F2FS, and NFS exhibit significant differences, simple statistics based directly on file names cannot objectively reflect their evolutionary commonalities. Therefore, this paper uses the general kernel contracts defined by the VFS (Virtual File System) as a baseline and classifies all code files participating in the evolution into five responsibility domains: Infrastructure, Metadata, Namespace, Data_IO, and Architecture_Specific.

Infrastructure represents the core infrastructure that handles file system mounting, unmounting, global initialization, and configuration parsing, with typical evidence including the `super.c` of each subsystem, main header files, and global state management logic. Metadata represents code responsible for managing the physical entities of files and their descriptors, covering the lifecycle of Inode objects, extended attributes, bitmaps, and access control (ACL) and other metadata consis-

tency logic. Namespace represents modules that maintain the logical directory tree structure and path navigation, mainly involving path resolution (Lookup), directory entry cache (Dentry), and file creation, deletion, and rename operations. Data_IO represents the data plane that handles byte stream exchange, including page read/write paths, Direct IO, page cache synchronization (Writeback), and related asynchronous I/O support logic. Architecture_Specific represents proprietary mechanisms introduced by each file system to achieve its specialized architectural objectives, such as the garbage collection (GC) and segment management of F2FS, the multi-block allocation algorithm (mballoc) of Ext4, the network protocol encapsulation (XDR) of NFS, and the decompression engine of EROFS.

Since a single commit often involves modifications to multiple source files, this paper adopts an "atomic expansion" strategy, treating the changes to each file in a patch as an independent evolution event. When a file name cannot be directly mapped to the above classification, the classifier combines the file's directory hierarchy, function prefixes (e.g., `f2fs_cp_` corresponding to Checkpoint), and code context to determine responsibility attribution. For general auxiliary code that cannot be clearly assigned to the above specialized domains, it is classified into the General_Common category as a supplement.

This responsibility attribution model provides a precise spatial baseline for subsequent bug latency analysis. By associating the defect lifecycles traced by the SZZ algorithm with specific responsibility domains, this paper can identify which functional modules exhibit higher defect concealment over the 10-year evolution period. For subsequent research, this paper will use this model to compare the stability gaps of different architectures under the same responsibilities and conduct in-depth code archaeology analysis on responsibility domains with anomalously long latency tails, in order to reveal the hidden technical debt in the process of modern storage adaptation.

F. Bug Latent Period

Bug latent period is defined as the time interval between the bug-introducing commit and the fixing commit. This paper preferentially uses the "Fixes:" tag in Linux commit messages for high-confidence tracing. If a fixing commit contains a tag in the form of `Fixes: <commit-hash>` ("commit subject"), the referenced fixed commit is extracted as the bug-introducing commit. Let the introduction time be T_{intro} and the fix time be T_{fix} , then the latent period is defined as:

$$\text{Latency} = T_{\text{fix}} - T_{\text{intro}} \quad (1)$$

Latent periods will be statistically analyzed by file system, bug type, and code responsibility respectively [2]. Analysis by file system can compare the speed at which defects are discovered in different systems; analysis by bug type can determine which types among semantic, concurrency, memory, and hardware interaction errors are more prone to long-term

latency; analysis by code responsibility can identify testing blind spots in exception paths, hardware-dependent paths, or concurrency paths.

For commits without a "Fixes:" tag but with high fix cost or significant research value, this paper adopts the SZZ approach for supplementary analysis: `git blame` is executed on the lines deleted or modified in the fixing patch to find the most recent commit that introduced or modified those lines, which serves as the candidate bug-introducing commit [2]. Since SZZ may produce misjudgments in cases of code movement, refactoring, and formatting changes, this paper will distinguish between high-confidence samples based on "Fixes:" and supplementary samples based on SZZ, avoiding the mixed interpretation of the two.

G. Fix Cost Measurement

Fix cost is measured by the modification scale of a patch. For each bug fix commit, this paper records the number of modified files, lines added, lines deleted, and total churn:

$$\text{churn} = \text{added_lines} + \text{deleted_lines} \quad (2)$$

where churn is the primary fix cost metric. This paper will calculate the mean, median, 90th percentile, and maximum values across the file system dimension and the bug type dimension respectively. The mean reflects the overall maintenance cost, the median reduces the influence of extreme commits, and the 90th percentile and maximum are used to identify high-cost tail samples. If the churn distribution of a certain bug type is notably higher, it indicates that such defects often involve broader state adjustments or architectural corrections.

In addition to statistical analysis, this paper will also conduct manual review of the bug fixes with the highest fix cost in each file system. The review focuses on: whether the bug spans multiple responsibility modules, whether it modifies core state machines or metadata invariants, whether it involves hardware constraints or protocol semantics, whether it introduces new locks, reference counts, or lifecycle rules, and what implications the fix offers for subsequent coding and review. Through this process, this paper aims to translate quantitative statistics into concrete engineering experience, for example, that error paths require explicit state verification, that page/folio lifecycle management requires strengthened reference count checks, that zoned device related code requires continuous verification of zone state and write pointers, and that NFS asynchronous RPC paths require focused checks on timeout, reconnection, and state consistency.

H. Threats to Validity

This paper faces the following threats to validity. First, patch type and bug fix identification rely on commit messages. Although Linux commits typically contain relatively sufficient context, there may still be cases of incomplete descriptions or inconspicuous keywords, leading to missed detection or misclassification. This paper reduces errors through positive keywords, negative filtering, and strong defect evidence rules,

and will estimate classification reliability through sampling and manual review in subsequent experiments.

Second, single-label classification simplifies defect causation. A bug may involve both concurrency and memory issues, or hardware error paths may trigger semantic inconsistency. This paper adopts single-label classification to ensure the interpretability of cross-file-system proportion comparisons, but the classification labels will not be interpreted as the sole root cause.

Third, path scope limitation may underestimate cross-layer defects. This paper only counts commits that directly modify files in the target file system directories, whereas certain file system defects may be fixed through modifications to VFS, memory management, block layer, or network stack common code. This choice improves the precision of file system attribution but sacrifices coverage of cross-layer fixes.

Finally, bug latent periods rely on the "Fixes:" tag or SZZ inference. The coverage of the "Fixes:" tag is incomplete, and SZZ may be affected by code movement, refactoring, and formatting changes. Therefore, latent period analysis will distinguish between high-confidence samples and supplementary inferred samples, and will avoid over-generalization when interpreting the results.

III. RESULT

A. Patch Type Classification Results

Based on the analysis of code evolution in F2FS, EXT4, ZONEFS, EROFS, and NFS from 2016 to April 2026, we counted security, feature, and maintenance changes at both the patch level and the commit level. Overall, the five filesystems contain 17,113 patches and 9,646 commits. Maintenance changes dominate across all filesystems, accounting for an average of 74.1% at the patch level and 77.9% at the commit level. Feature changes rank second, accounting for 19.4% at the patch level and 14.0% at the commit level. Security changes have the lowest proportion, with averages of 6.5% at the patch level and 8.1% at the commit level. We also observe that feature-related changes tend to modify more files on average than the other two types of changes [3].

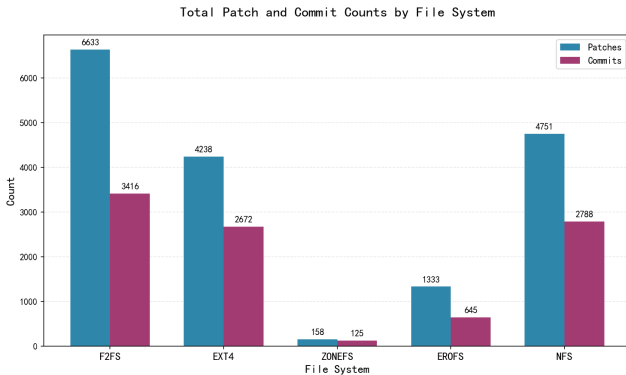


Fig. 1: Overall Distribution of Patch and Commit Categories Across Five Filesystems

From the distributional perspective, the proportion of security-related changes varies significantly among filesystems. EXT4 has the highest security-related proportion, with security patches accounting for 18.0% and security commits accounting for 21.2%. This is substantially higher than the other four filesystems. This phenomenon may reflect the fact that EXT4, as one of the most widely used filesystems in the Linux ecosystem, has undergone a longer process of security auditing and vulnerability fixing. Its large user base and broad deployment also make security issues more likely to be discovered and reported. By contrast, ZONEFS has the lowest security-related proportion, with only 0.6% of patches and 0.8% of commits related to security. ZONEFS is a relatively new filesystem specifically designed for zoned storage devices [4]. Its codebase is small, its functionality is focused, and its attack surface is limited, which may explain the small number of exposed security issues. F2FS, EROFS, and NFS fall in the middle range, with security-related proportions between 5.0% and 6.4%, indicating a relatively similar level of security maintenance effort.

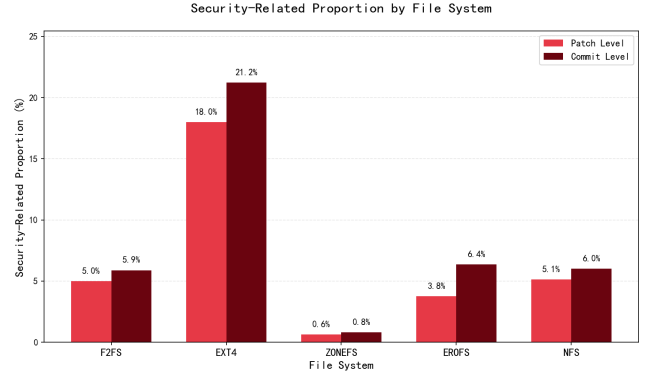


Fig. 2: Comparison of Security-Related Change Proportions Across Filesystems

The distribution of feature-related changes presents a different pattern from security changes. EROFS is the most prominent filesystem in feature-related changes, with 28.0% at the patch level and 18.8% at the commit level, both the highest among the five filesystems. As a read-only compressed filesystem mainly used in Android systems and embedded scenarios [5], the high feature-development proportion of EROFS may be related to its rapid development stage, during which new compression algorithms, optimization strategies, and usage scenarios are continuously added. ZONEFS also shows active feature development, with 22.2% of patches and 18.4% of commits related to features [4]. This is consistent with its status as an emerging filesystem that still needs to improve basic functionality and adapt to new hardware. F2FS has 18.3% feature-related patches and 11.8% feature-related commits, indicating a relatively balanced pace of feature evolution. EXT4 has a relatively low feature-related proportion, with 12.0% at the patch level and 8.9% at the commit level, reflecting the characteristics of a mature and stability-oriented

filesystem whose feature iteration has entered a slower stage. NFS is in the middle range in terms of feature-related changes.

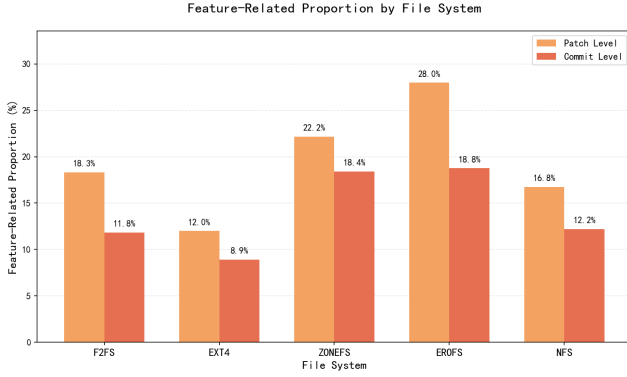


Fig. 3: Comparison of Feature-Related Change Proportions Across Filesystems

Maintenance changes account for the largest proportion in all filesystems, reflecting the central role of code maintenance in software evolution. NFS has the highest maintenance-related patch proportion, reaching 78.1%, which reflects continuous investment in compatibility, performance optimization, and bug fixing for a network filesystem. F2FS has the highest maintenance-related commit proportion, reaching 82.3%, indicating that this filesystem has undergone substantial code cleanup, refactoring, and general bug fixing. ZONEFS also has high maintenance-related proportions, with 77.2% at the patch level and 80.8% at the commit level. EXT4 has the lowest maintenance-related proportion among the five filesystems, but it still exceeds 70.0%, showing that even for a mature codebase, maintenance work remains the dominant activity.

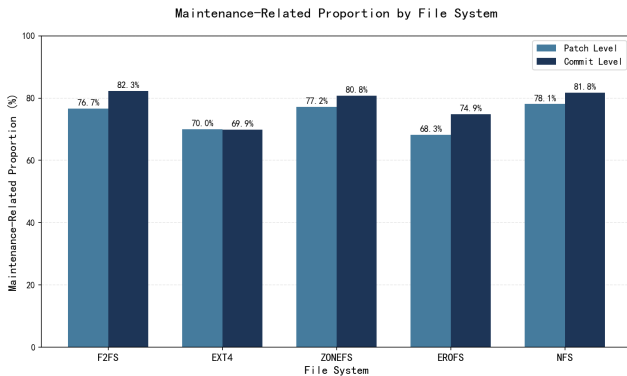


Fig. 4: Comparison of Maintenance-Related Change Proportions Across Filesystems

The difference between patch-level and commit-level proportions is also noteworthy. Feature-related changes account for an average of 19.4% at the patch level but only 14.0% at the commit level. This indicates that feature changes often involve modifications to multiple files, meaning that a feature commit usually contains multiple file-level patches. In contrast, security-related changes account for 6.5% at the patch level

and 8.1% at the commit level, suggesting that security commits usually contain fewer patches and are often concentrated in a small number of files or even a single file. The proportions of maintenance changes are the closest between the two levels, indicating that the file modification scope of maintenance operations is relatively consistent with commit granularity.

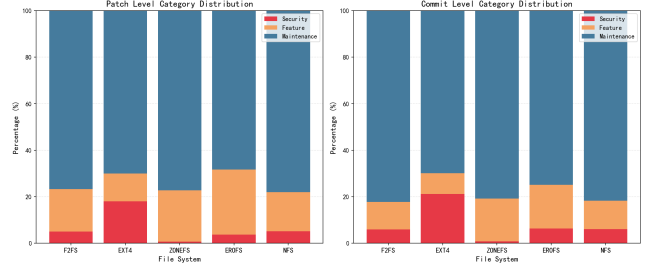


Fig. 5: Comparison Between Patch-Level and Commit-Level Classification Proportions

Overall, the five filesystems show different code evolution characteristics. EXT4 exhibits typical characteristics of a mature filesystem, with a high security-maintenance proportion and a low feature-development proportion. EROFS and ZONEFS show characteristics of emerging filesystems, with active feature development and limited accumulated security issues. F2FS and NFS lie between these two groups: their feature development continues but is not overly aggressive, while maintenance work occupies most development resources. These distributional characteristics are closely related to each filesystem's development stage, application scenario, and code scale. The following time-dimensional statistical analysis further explains the evolution and development of these five filesystems over the past decade.

Based on annual patch-level statistics for F2FS, EXT4, ZONEFS, EROFS, and NFS from 2016 to 2026, the five filesystems show clearly different evolutionary characteristics over time.

In terms of overall development activity, F2FS has the largest number of patches, with 6,633 patches over 11 years. Its annual patch count peaked at 918 in 2016 and then showed a fluctuating downward trend, but it still had 866 patches in 2025, indicating sustained development activity. EXT4 has 4,238 patches and ranks second by this measure. Its development activity is relatively stable, with annual patch counts fluctuating between 300 and 600 and reaching a peak of 575 in 2020. NFS has 4,751 patches, also placing it among the most active filesystems; it had 677 patches in 2016 and then showed a downward trend. EROFS, as a newer filesystem, has shown steady growth in development activity since its appearance in 2019, reaching a peak of 224 patches in 2023. ZONEFS has the fewest patches, with only 158, which is consistent with its role as a specialized filesystem.

Regarding the evolution of security-related patches, EXT4 shows the most prominent characteristics. The proportion of security-related patches in EXT4 reached a peak of 32.1% in 2018, and its annual average was as high as 22.3%, far

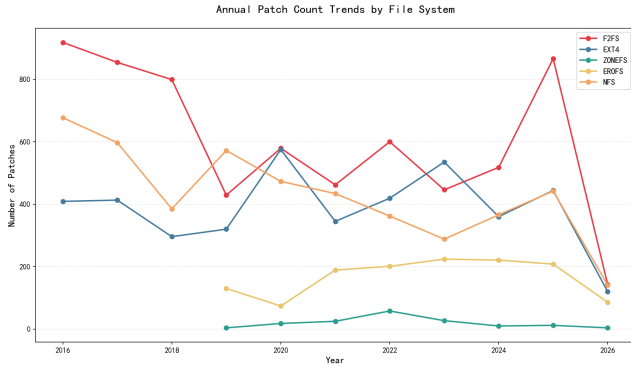


Fig. 6: Annual Patch Count Changes Across Filesystems

exceeding the other filesystems. This phenomenon is closely related to EXT4's position as one of the most widely used filesystems in the Linux ecosystem. Its large codebase and user base make security issues easier to discover and fix. It is worth noting that although the security-related proportion of EXT4 declined after 2018, it remained between 12% and 21%, reflecting the continuous security-maintenance demand of a mature filesystem.

The proportion of security-related patches in F2FS reached a peak of 10.6% in 2018 and then generally declined, remaining between 3% and 8% in recent years, with an annual average of 5.9%. The security-related proportion of EROFS gradually increased from 0% in 2019 to 5.8% in 2026, indicating that security attention has increased as the filesystem has matured. The security-related proportion of NFS is relatively stable, remaining between 4% and 9%, with an annual average of 5.9%. ZONEFS has very few security-related patches, with only one security-related patch over 11 years, which is consistent with its small codebase and focused functionality.

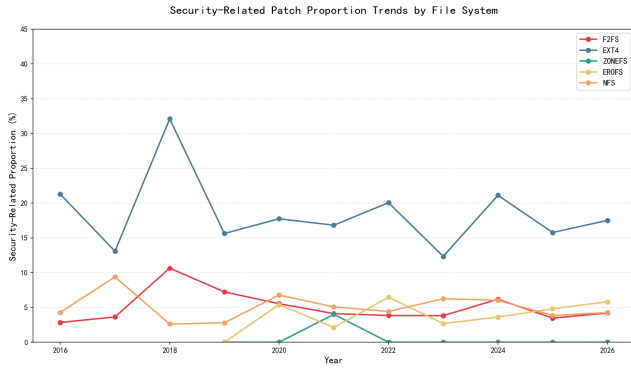


Fig. 7: Annual Trend of Security-Related Patches

The evolution of feature-related patches reveals the development stage of each filesystem. EROFS is the most active in feature development, with the proportion of feature-related patches reaching 52.9% in 2021 and an annual average of 17.6%. As a read-only compressed filesystem mainly used in Android and embedded scenarios, its high feature-development proportion indicates that EROFS is still in a rapid development

stage, continuously adding new compression algorithms, optimization strategies, and usage-scenario support.

ZONEFS had a feature-related proportion of 100% when it first appeared in 2019 because it was newly developed. However, this proportion declined rapidly afterward, and feature development slowed significantly after 2022, indicating that the filesystem entered a maintenance stage after its basic framework was established. The feature-related proportion of F2FS reached a peak of 32.9% in 2021 and then decreased, remaining between 9% and 27% in recent years. Its annual average was 19.4%, showing a continuous but not overly aggressive feature-evolution pace. The feature-related proportion of NFS reached a peak of 23.0% in 2024, but sharply dropped to 1.4% in 2026, which may be related to the fact that the 2026 data only covers up to April. EXT4 has the lowest feature-development proportion, with an annual average of only 9.0%, reflecting that feature iteration in this mature filesystem has entered a slow stage.

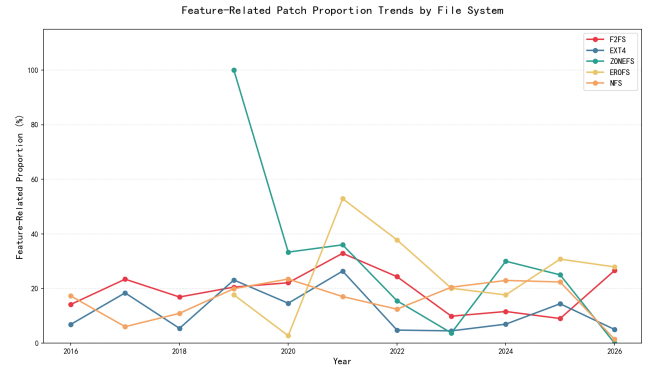


Fig. 8: Annual Trend of Feature-Related Patches

Maintenance-related patches dominate all filesystems, demonstrating the core role of code maintenance in software evolution. The maintenance-related proportion of NFS increased from 78.4% in 2016 to 94.3% in 2026, showing an evident upward trend and indicating that the filesystem has entered a stable stage dominated by maintenance. The maintenance-related proportion of F2FS reached a peak of 88.6% in 2025, with an annual average of 74.1%. EXT4 reached a maintenance-related peak of 83.2% in 2023, with an annual average of 71.9%. The maintenance-related proportion of EROFS decreased from 82.3% in 2019 to 45.0% in 2021 and then rebounded to around 66%, reflecting a typical pattern in which the maintenance proportion decreases during intensive feature development and rises again after major features are completed.

Through the overall time-dimensional analysis, the five filesystems can be divided into three development stages. EXT4 is in a mature and stable stage, with a high security-maintenance proportion, a low feature-development proportion, and stable development activity. EROFS and ZONEFS are in a rapid development stage, with active feature development, gradually increasing security accumulation, and still-expanding codebases. F2FS and NFS are in a mature

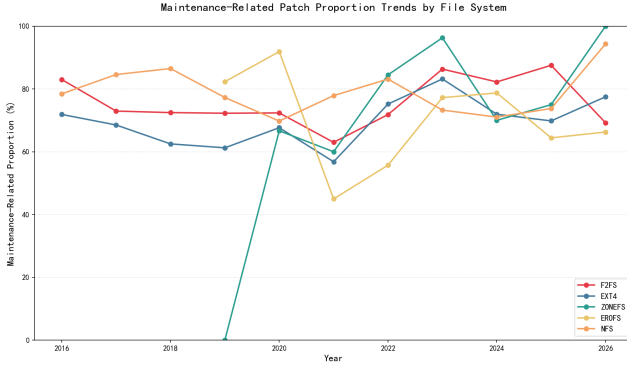


Fig. 9: Annual Trend of Maintenance-Related Patches

evolution stage, with continuous but not overly aggressive feature development, maintenance work as the dominant activity, and development activity showing a fluctuating downward trend. These differences in evolution are closely related to each filesystem's birth time, application scenario, code scale, and community activity, providing quantitative evidence for understanding the long-term evolution patterns of filesystems.

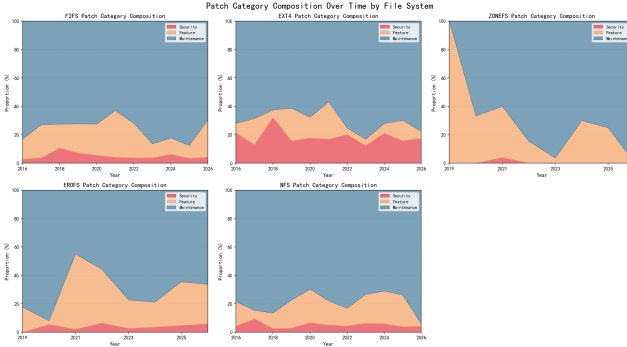


Fig. 10: Comprehensive Comparison of Time-Dimensional Evolution Characteristics Across Five Filesystems

B. Bug Fix Identification and Classification Results

This section presents a statistical analysis of bug-fix commits in the five filesystems from January 2016 to April 2026, based on the completed bug-fix identification and four-class bug classification results. Overall, 3,906 "bug-fix commit-filesystem" samples were identified. For comparison, Table I reports the absolute number of each bug type in each filesystem, and Table II further reports the normalized proportion within each filesystem.

TABLE I: Number of Bug Types in Each Filesystem

Filesystem	Bug Fixes	Semantic	Concurrency	Memory	Hardware-Interaction
ext4	1171	381	187	353	250
F2FS	1382	548	274	285	275
ZoneFS	34	9	4	2	19
EROFS	184	57	12	50	65
NFS	1135	374	218	243	300

Tables I and II, together with Fig. 11 to Fig. 13, show that the bug-type structures of different filesystems differ signifi-

TABLE II: Normalized Proportion of Bug Types in Each Filesystem

Filesystem	Semantic	Concurrency	Memory	Hardware-Interaction
ext4	32.54%	15.97%	30.15%	21.35%
F2FS	39.65%	19.83%	20.62%	19.90%
ZoneFS	26.47%	11.76%	5.88%	55.88%
EROFS	30.98%	6.52%	27.17%	35.33%
NFS	32.95%	19.21%	21.41%	26.43%

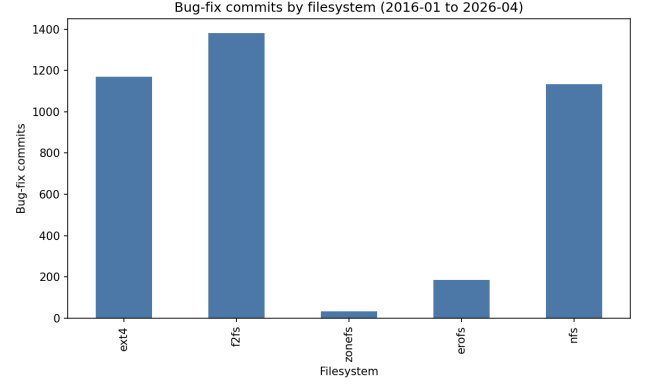


Fig. 11: Comparison of Bug Type Counts Across Filesystems

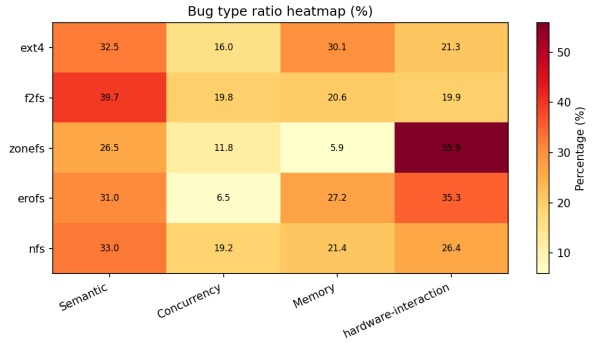


Fig. 12: Comparison of Normalized Bug Type Proportions Across Filesystems

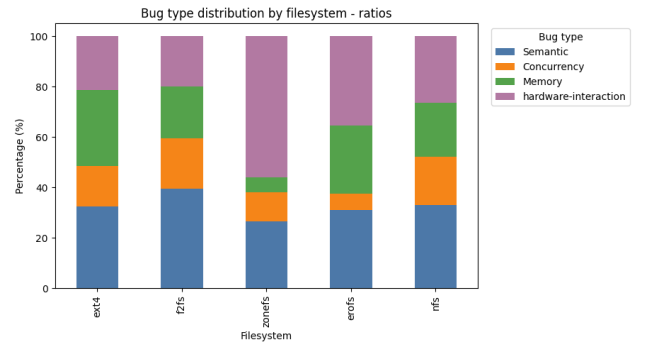


Fig. 13: Structural Distribution of Bug Types Across Five Filesystems

cantly. The following analysis discusses the four dimensions: Semantic, Concurrency, Memory, and Hardware-Interaction.

1) *Semantic Bug: Defects Dominated by State Machines and Metadata Semantics*: Semantic bugs are the most common type among the five filesystems. In ext4, F2FS, EROFS, and NFS, Semantic bugs all account for a high proportion. F2FS has the highest proportion at 39.65%, followed by NFS at 32.95%, ext4 at 32.54%, and EROFS at 30.98%. ZoneFS has a Semantic proportion of 26.47%, which is lower than the other filesystems but still represents its second-largest bug type.

The essence of a Semantic bug is inconsistency between internal filesystem logic and expected semantics. Such problems are usually related to filesystem state machines, metadata invariants, and error-path semantics. Typical examples include incorrect inode state update order, inconsistent extent mappings, incomplete checkpoint state checks, incorrect error-code propagation, missing rollback on exceptional paths, and directory-entry or xattr handling that does not match expectations [6].

The high proportion of Semantic bugs in F2FS mainly comes from its complex Flash-oriented internal state management [7]. F2FS builds its write and reclamation strategy around mechanisms such as segments, sections, checkpoints, node/data separation, hot and cold data classification, garbage collection, and discard. These mechanisms introduce many cross-module state dependencies: a single write may affect segment state, checkpoint validity, garbage-collection strategy, dirty-page writeback, and space reclamation. If a state check misses a boundary condition, or if an error path fails to update metadata consistently, a Semantic bug may occur. Therefore, the high Semantic proportion in F2FS reflects not isolated programming mistakes but the enlarged state space introduced by Flash-specialized design.

The Semantic bug proportion in ext4 is also high. As a mature general-purpose filesystem, ext4's complexity mainly comes from metadata semantics and compatibility paths accumulated over long-term evolution [8]. ext4 must maintain mechanisms such as inodes, extents, directory entries, journaling, quotas, orphan lists, delayed allocation, fsync, and recovery. Unlike F2FS, the semantic complexity of ext4 mainly comes from consistency constraints that a general-purpose filesystem must preserve over time. For example, some fixes may involve only a small amount of code, but the underlying issue may concern whether metadata remains consistent after crash recovery, whether an error path correctly releases an orphan inode, or whether fsync satisfies persistence semantics. Thus, Semantic bugs in ext4 reflect the maintenance difficulty caused by the interaction of historical features and consistency semantics in a mature filesystem [9].

NFS has a Semantic bug proportion of 32.95%, and its sources differ from those of local filesystems. The NFS client must map local VFS semantics to remote server protocol semantics. Attribute caching, delegation, layout, open state, lock state, revalidation, rename, and close-to-open consistency may all introduce semantic inconsistencies. Semantic bugs in

NFS are often not simple local metadata errors; rather, they arise from incomplete understanding of remote state by the client or failure to maintain correct local semantics under network exceptions, reconnections, timeouts, and server-returned errors. Therefore, Semantic bugs in NFS reflect the semantic gap between distributed state and local file abstractions [10].

Semantic bugs in EROFS mainly come from read-only compressed formats and metadata parsing [5]. Because EROFS does not support general write paths, its state space is smaller than that of ext4 and F2FS. However, it must correctly interpret on-disk formats, compressed-block indexes, directory structures, and data mappings in the page cache. Incorrect handling of compressed layouts, block boundaries, or metadata parsing conditions may lead to read errors or data inconsistency. ZoneFS has a relatively low Semantic bug proportion because its design is thin and its functional target is focused [4], with fewer traditional metadata state machines. Nevertheless, it still needs to maintain the relationship among file size, zone state, write position, and VFS semantics, so a certain proportion of semantic errors remains.

Overall, the distribution of Semantic bugs indicates that filesystems relying more heavily on complex state machines, metadata invariants, and cross-layer semantic mappings are more prone to semantic defects. The high proportion in F2FS corresponds to Flash-specialized state machines, ext4 corresponds to mature metadata semantics, NFS corresponds to remote protocol semantics, EROFS corresponds to compressed-format semantics, and ZoneFS has a relatively low proportion due to its narrower design.

2) *Concurrency Bug: Defects Caused by Asynchronous Paths and Shared State*: Concurrency bug proportions differ noticeably among the five filesystems. F2FS has the highest proportion at 19.83%, followed by NFS at 19.21%, ext4 at 15.97%, ZoneFS at 11.76%, and EROFS at only 6.52%. This indicates that concurrency errors are more concentrated in filesystems with complex asynchronous execution paths, background threads, or remote-request states.

F2FS has the highest Concurrency bug proportion because it contains multiple parallel internal maintenance paths. F2FS must maintain consistency among foreground writes, background garbage collection, checkpoints, flushes, discards, dirty-page writeback, and segment reclamation. The garbage-collection thread may race with normal writes for space-allocation state; checkpointing may race with writeback paths for metadata state; and discard and flush may be related to device submission ordering. Because these paths often span multiple locks and asynchronous tasks, omissions in lock granularity, execution ordering, or state checking may cause race conditions, deadlocks, or inconsistent states. Therefore, concurrency bugs in F2FS result from the combined effects of high-performance Flash adaptation and background cleaning mechanisms.

NFS also has a high Concurrency bug proportion. Unlike local filesystems, NFS concurrency complexity comes from the combination of local multi-process access and asynchronous remote RPC execution. The NFS client must handle

concurrent reads and writes, asynchronous RPC callbacks, retransmissions, timeouts, reconnections, delegation recall, lock-state recovery, and page-cache writeback. A request may appear complete locally, while the remote server state, network transport state, or cache state is still changing. The resulting shared states include inode attribute caches, open owners, lock owners, sessions, layouts, and transport connections. Improper synchronization of these states can easily lead to races and ordering errors.

The Concurrency bug proportion of ext4 is lower than F2FS and NFS, but it still reaches 15.97%. As a mature local filesystem, ext4's main concurrency risks come from journaling, delayed allocation, writeback, truncate, rename, extent-tree updates, and quotas. Compared with F2FS, ext4 has lower background-strategy complexity; compared with NFS, it has no remote-request state. However, due to its broad deployment, numerous execution paths, and complex historical code, ext4 still has a certain number of concurrency defects.

EROFS has the lowest Concurrency bug proportion. The main reason is that EROFS is a read-only filesystem and does not involve general writes, space allocation, transaction commits, or complex recovery paths. This naturally reduces foreground/background write competition and metadata-update races. Although reads, readahead, decompression, and page-cache paths may still execute concurrently, the amount of shared mutable state is significantly smaller than in ext4 and F2FS, leading to a lower concurrency bug proportion.

ZoneFS also has a relatively low Concurrency bug proportion, though it is higher than EROFS. Its small codebase and thin functionality reduce complex lock interactions. However, ZoneFS still needs to handle sequential write positions, file sizes, and zone-state updates, which may experience ordering problems under concurrent writes or truncate operations. Therefore, ZoneFS has some concurrency risk, but it is not the dominant issue.

Overall, the proportion of concurrency bugs is strongly related to the amount of shared mutable state and the number of asynchronous paths in a filesystem. F2FS and NFS have high proportions, corresponding respectively to background cleaning/reclamation mechanisms and asynchronous RPC state machines. ext4 remains at a medium level due to mature multi-path writes and journaling. EROFS and ZoneFS have lower proportions because they contain less mutable state or have thinner designs.

3) *Memory Bug: Defects in Object Lifetimes and Cache Paths:* Memory bugs are relatively prominent in ext4 and EROFS. ext4 has a Memory bug proportion of 30.15%, making it its second-largest bug type and close to Semantic bugs. EROFS has 27.17%, NFS has 21.41%, F2FS has 20.62%, and ZoneFS has only 5.88%. This distribution shows that memory-related problems do not simply scale with code size, but are related to internal object lifetimes, cache paths, and data-structure complexity.

ext4 has the highest Memory bug proportion mainly because it manages a large number of long-lived and mutually referenced kernel objects. ext4 involves inodes, extent status

trees, buffer heads, journal handles, quota objects, orphan inodes, page cache, and delayed-allocation structures. These objects have different lifetimes on normal paths, error paths, and recovery paths. Some objects are held by the VFS, some are indirectly referenced by the journal or block layer, and others are related to page-cache or writeback state. If reference counts, release ordering, or error-path cleanup are incomplete, memory leaks, null-pointer dereferences, or use-after-free bugs can occur. Therefore, the high Memory bug proportion of ext4 reflects the lifetime-management difficulty caused by mature and complex data structures and intertwined historical paths.

EROFS also has a high Memory bug proportion, which is closely related to its compressed read path. The core logic of EROFS includes compressed-block indexes, decompression buffers, page/folio management, readahead, and page-cache mapping. A compressed filesystem must map logical pages, physical compressed blocks, decompressed output pages, and cached pages correctly. One physical compressed block may serve multiple logical pages, and multiple read requests may share or compete for decompression resources. These paths are prone to page-reference errors, buffer-boundary errors, temporary-object release problems, and error-path cleanup problems. Since EROFS does not involve complex writes, its memory problems are mainly concentrated in object lifetimes during reading and decompression.

NFS and F2FS have medium Memory bug proportions. Memory problems in NFS are usually related to RPC request objects, page cache, attribute caches, state owners, layouts, and transport-related objects. Network exceptions and retries may extend object lifetimes and make release timing more complex. Memory problems in F2FS are related to node/page structures, segment information, checkpoint data, garbage-collection paths, and bio/page management. Both filesystems have complex object management, but in the overall bug structure, protocol semantics and concurrency issues are more prominent in NFS, while semantic state issues are more prominent in F2FS. Thus, their Memory bug proportions are lower than ext4.

ZoneFS has the lowest Memory bug proportion, only 5.88%. This is mainly due to its minimalist design. The goal of ZoneFS is not to implement complex metadata management or compressed caching, but to expose zones of a zoned block device as filesystem objects. Its internal data structures are fewer, and object lifetimes are relatively direct, reducing opportunities for memory leaks, use-after-free bugs, and complex reference-counting errors. However, since ZoneFS has only 34 samples, its proportion is affected by a small number of commits and should not be overinterpreted as evidence that it has no memory risk.

Overall, the proportion of Memory bugs is closely related to the complexity of the filesystem's internal object graph, cache paths, and error-path cleanup. ext4 and EROFS represent two typical sources: ext4 reflects long-accumulated data structures and reference relationships in a mature general-purpose filesystem, while EROFS reflects page/folio and buffer lifetime management in compressed read paths.

4) *Hardware-Interaction Bug: Defects Caused by Hardware Constraints and External Boundaries*: Hardware-Interaction bugs are the bug type with the most significant differences among the five filesystems. ZoneFS has the highest proportion, reaching 55.88%; EROFS has 35.33%; NFS has 26.43%; ext4 has 21.35%; and F2FS has 19.90%. This result shows that when a filesystem’s core design goal is directly related to underlying hardware or external protocol boundaries, Hardware-Interaction bugs increase significantly.

ZoneFS has a Hardware-Interaction bug proportion of more than half, which is the most expected result. ZoneFS itself is built around zoned block devices. Its core task is not to hide hardware constraints, but to expose zone semantics such as sequential write rules, zone size, write pointer, reset, and open/close state through a file interface. Compared with traditional filesystems, the correctness of ZoneFS highly depends on underlying device state. If the filesystem incorrectly determines zone state, mishandles the write pointer, insufficiently validates sequential write constraints, or fails to synchronize device state correctly during reset, append, or truncate operations, a hardware-interaction error may occur. Therefore, the bug distribution of ZoneFS reflects the design characteristic of a “thin filesystem plus strong hardware constraints”: traditional semantic and memory complexity is low, while device-semantics adaptation becomes the main source of defects.

EROFS has a Hardware-Interaction bug proportion of 35.33%, second only to ZoneFS. Although EROFS is not a filesystem that directly exposes special hardware constraints, it strongly depends on the read path, page cache, readahead, block mapping, and compressed-block layout. A core challenge of a compressed read-only filesystem is to map on-disk compressed data blocks efficiently and correctly to memory pages and user read requests. I/O boundaries, block alignment, compressed cluster size, readahead strategy, and error handling may all lead to hardware-interaction problems. In other words, Hardware-Interaction bugs in EROFS mainly come not from “device write rules”, but from the mapping between compressed layouts and I/O read paths.

NFS has a Hardware-Interaction bug proportion of 26.43%. In this context, “hardware interaction” should be understood more broadly as interaction with external system boundaries. The key boundary of the NFS client is not the local block device, but the remote server and network transport layer. RPC, XDR, transport, sockets, timeouts, RDMA, and server state can all affect filesystem semantics. Therefore, Hardware-Interaction bugs in NFS actually reflect defects in the interaction between the filesystem and network protocol endpoints. For example, improper handling of server-returned states, incorrect RPC timeout and retry logic, or inconsistent state during transport reconnection may all fall into this category.

The Hardware-Interaction bug proportions of ext4 and F2FS are relatively low, at 21.35% and 19.90%, respectively. This does not mean that they do not care about hardware. Rather, their main complexity lies more in internal filesystem state and metadata management. ext4 accesses underlying storage

through mature block-device abstractions, and hardware differences are usually hidden by the block layer, so hardware-interaction problems are not dominant. Although F2FS is designed for Flash/SSD devices, its bugs more often appear as semantic problems in segment, checkpoint, garbage-collection, and discard strategies, rather than direct device-interaction errors. This shows that hardware-oriented optimization does not necessarily lead to the highest Hardware-Interaction bug proportion. If hardware characteristics are absorbed into internal strategy state machines, the resulting bugs may appear more often as Semantic bugs.

Overall, the proportion of Hardware-Interaction bugs depends on how a filesystem is coupled with external boundaries. ZoneFS directly exposes device constraints to filesystem semantics, so this type has the highest proportion. EROFS strongly depends on I/O mapping in compressed read paths, so its proportion is also high. The external boundary of NFS is the network protocol and remote server, making this category prominent as well. Although ext4 and F2FS both run on block devices, more of their defects are absorbed by internal state machines and metadata logic.

C. Architectural Responsibility Attribution and Bug Lifespan Analysis

In terms of the spatial distribution of code responsibilities, we first conducted an attribution analysis of the code assets for the VFS (Virtual File System) core framework and five representative filesystems. This methodology builds upon the foundational framework of studying filesystem evolution through patch classification [8]. This analysis reveals significant disparities in design priorities across different responsibility domains.

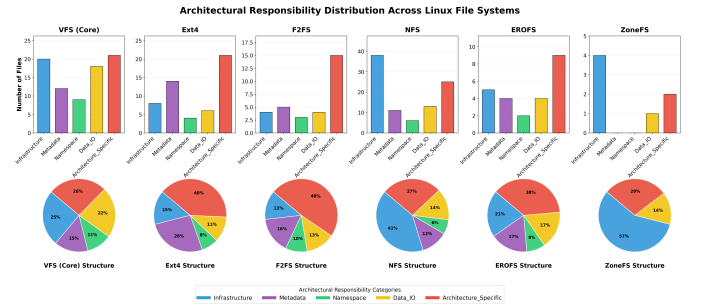


Fig. 14: Architectural Responsibility Distribution Across Linux File Systems.

As illustrated in Fig. 14, **Ext4**, as a mature and general-purpose journaling filesystem, exhibits a file distribution pattern across *Infrastructure*, *Metadata*, *Namespace*, and the data plane that is highly consistent with the VFS core framework. This demonstrates its architectural design’s rigorous adherence to standard Linux kernel contracts.

In contrast, **F2FS** and **NFS** show a pronounced “mechanism-heavy” tendency, with their *Architecture_Specific* code accounting for 48.4% and 26.9% of their respective totals. These quantitative results reveal that flash-specific

logic [7] and complex distributed network protocols occupy the dominant implementation space within kernel-level engineering. Notably, the architectural distribution of **ZoneFS** is characterized by extreme skewness; 57.1% of its files are concentrated in the *Infrastructure* domain, while independent implementations for *Metadata* and *Namespace* are entirely absent. This “architectural transparency” design philosophy, which aligns with the requirements of Zoned Storage Devices [11], is empirically confirmed by the data, which shows that ZoneFS minimizes kernel-side maintenance overhead through “logic subtraction”—directly mapping files to heterogeneous hardware (such as ZNS SSDs) and thereby bypassing the code bloat typically found in the intermediate layers of traditional filesystems.

Following the characterization of the spatial distribution of code responsibilities, we further introduce the *Bug Lifespan (Latency)* metric. To ensure high-fidelity calculation of this temporal dimension, we adopt the **SZZ algorithm** [12] to trace fix-inducing changes back to their origins, allowing us to detect the distribution patterns of implicit technical debt across various subsystems.

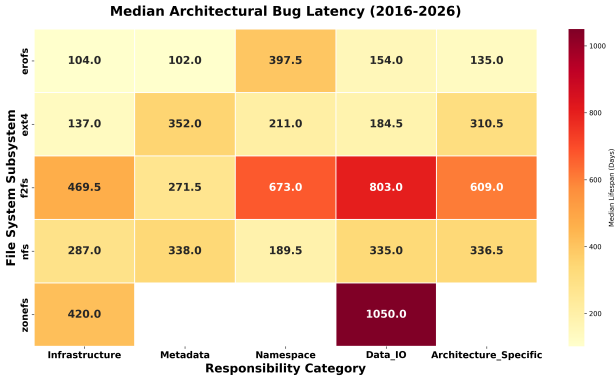


Fig. 15: Median Architectural Bug Latency Across Linux File Systems (2016-2026).

The median latency heatmap reveals a strong positive correlation between bug stealthiness and the depth of architectural compensation. As illustrated in Fig. 15, **F2FS** exhibits the highest median lifespans across the board, particularly in the *Data_IO* (803.0 days) and *Namespace* (673.0 days) domains. This suggests that the complex Copy-on-Write (CoW) paths and associated metadata synchronization logic in F2FS accumulate deep “hibernating” defects, which typically manifest only after the code has been in the repository for more than two years, a common characteristic of latent bugs in complex system software [13].

Another significant discovery is that **ZoneFS** maintains a median bug lifespan of 1050.0 days in the *Data_IO* sector. Despite the minimal code scale of this subsystem, such extreme latency highlights the profound verification challenges inherent in handling hardware-level boundary anomalies.

In contrast, **EROFS** demonstrates relatively low median lifespans across all dimensions, with particularly high convergence in *Infrastructure* and *Metadata*. This empirically

validates the advantages of read-only architectures [14] in reducing system state-space complexity and accelerating defect exposure.

To further analyze the dispersion characteristics of defect distributions, we compared the median latency with the mean bug lifespan heatmap to capture the impact of long-tail defects on overall system stability.

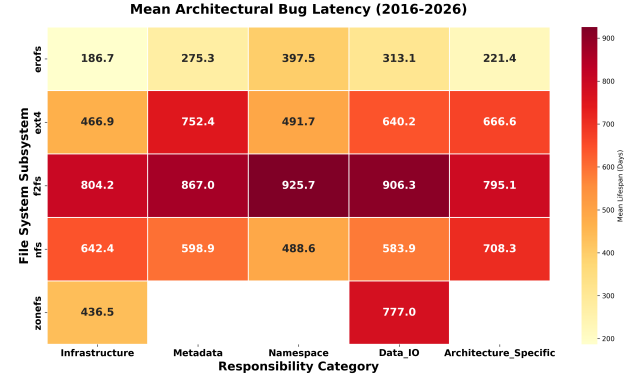


Fig. 16: Mean Architectural Bug Latency Across Linux File Systems (2016-2026).

A comparative analysis of the median and mean heatmaps shows that the mean lifespan for all subsystems is consistently and significantly higher than the median. This provides statistical evidence of a pronounced “long-tail distribution” of defects within the Linux storage stack, a phenomenon that aligns with previous observations that a small fraction of files often accounts for the majority of late-discovered faults in large-scale software [15].

Specifically, in the **Ext4 Metadata** domain, the mean (752.4 days) significantly deviates from the median (352.0 days), indicating that even in highly mature modules, a small number of deeply embedded structural defects with extremely long latencies persist. This observation reinforces the theory that architectural maturity does not inherently eliminate latent risks, as complex interactions can keep certain bugs dormant for years [16]. Meanwhile, **F2FS** maintains a high global mean latency (ranging from 800 to 900 days) across all domains, further solidifying its empirical characterization as a “high-maintenance-cost system.”

Through the cross-validation of these two heatmaps, we identified the full responsibility domains of **F2FS** and the data plane path of **ZoneFS** as the primary targets for our subsequent fine-grained code archaeology. These high-latency hotspots not only quantify the “stability premium” paid by specialized storage systems during their evolution but also provide a definitive “risk map” for future targeted troubleshooting of implicit concurrency conflicts and state-machine violations, which are known to be the most challenging categories of bugs to detect in the Linux kernel [17].

To achieve a more refined stability assessment, this study introduces the *Hotspot Score* model. We define the score for

an individual file f as the product of its median bug lifespan and its total bug count:

$$H_f = \tilde{L}_f \cdot N_f \quad (3)$$

where H_f is the *Hotspot Score* of file f , $\tilde{L}_f$ is its *median bug lifespan*, and N_f is the *total count of bugs* identified in that file [18].

1) *Cross-System Hotspot Distribution Characteristics*: The Hotspot Score reveals significant generational gaps in risk distribution across different filesystems. As shown in Fig. 17, the core files of **F2FS** (e.g., *file.c*, *f2fs.h*, and *data.c*) achieved scores as high as 59,823.0, exceeding those of **EROFS** and **ZoneFS** by an order of magnitude. This provides strong empirical evidence that **F2FS**’s core logic layer has become a major center for technical debt during the pursuit of high-performance flash adaptation [7]. In contrast, **Ext4** hotspots are highly concentrated in *mballoc.c* (multi-block allocator) and *inode.c*, suggesting that even in a mature system evolving for over two decades, the complexity of physical block layout algorithms remains a “deep water zone” for defect management, consistent with findings that allocator logic is a perennial source of OS bugs [8].

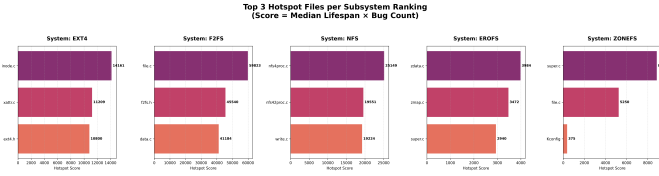


Fig. 17: Top 3 Hotspot Files per Subsystem Ranking based on Hotspot Score.

2) Global Stability Dynamics: Bug Survival Analysis:

To quantify the efficiency of stability convergence across systems from a macroscopic dimension, we introduce the *Bug Survival Curve* (Cumulative Distribution Function, CDF). This methodology aligns with recent empirical software engineering studies that utilize survival analysis to evaluate the “clean-up” efficiency of open-source projects [19].

The curve illustrates the “purification speed” of latent defects over time. Experimental observations reveal a distinct stability convergence gradient across architectures. As shown in Fig. 18, **EROFS** (purple curve) exhibits the highest self-healing efficiency, clearing approximately 80% of defects within 300 days of their introduction—well before the one-year baseline. This quantifies the inherent advantage of read-only, compressed architectures in accelerating defect exposure due to their restricted state space. [14].

Conversely, the **F2FS** curve (blue line) shows a significant “rightward shift.” At the one-year mark (365 days), its defect resolution proportion is only approximately 40%, which is the lowest among all systems and represents a twofold efficiency gap compared to **EROFS**. Its low slope in the mid-to-long term demonstrates that complex “software-defined hardware compensation logic” creates a deep shielding effect for defects.

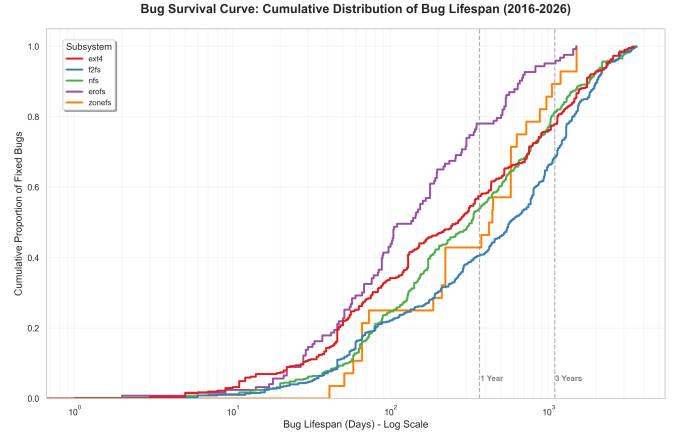


Fig. 18: Cumulative Distribution Function (CDF) of Bug Lifespan across Subsystems (2016–2026).

Notably, **ZoneFS** (orange curve) exhibits a unique “step-like evolution” pattern. The prolonged horizontal plateaus between 50 and 400 days validate our hypothesis of “logic hibernation” in minimalist architectures that rely heavily on rare hardware error feedback, a side effect of moving complexity from the filesystem to the device or application layer [11]. However, after crossing the 1,000-day threshold, the resolution rate of **ZoneFS** rapidly surpasses all other filesystems except **EROFS**. This suggests that in the late stages of evolution, the maintenance burden of minimalist architectures is significantly lower than that of systems requiring persistent optimization.

3) Qualitative Attribution of Long-latency Defects:

Through an in-depth code walkthrough of 39 extreme samples with lifespans exceeding 1,000 days, we summarize three typical defect induction patterns:

- **Multidimensional Asynchronous State Machine Conflicts**: The most extreme case identified is a defect in **F2FS** *node.c* (Commit 7633a73), with a lifespan of 3,552 days. This bug involves a flag inconsistency between the atomic commit and checkpoint mechanisms under concurrent states. Such defects prove that complex asynchronous mechanisms designed to compensate for hardware limitations can produce non-deterministic races over extremely long timelines [17].
- **Logic Hibernation in Error Paths**: In **ZoneFS** *file.c*, we identified a median lifespan of 1,050 days. These defects typically reside in specific exception-handling paths triggered only by non-standard hardware responses (e.g., Zone corruption). This “low-frequency trigger” characteristic allows defects to remain dormant until large-scale deployment of new hardware exposes the edge cases [9].
- **Subtle Architectural Contract Violations**: Long-period patches in **Ext4** *mballoc.c* (e.g., Commit 496bb99) often involve violations of general kernel memory management contracts, such as *might_sleep()* constraints. These bugs, with a median lifespan exceeding 1,700 days, reflect architectural erosion where low-level implementations fail

to align with subtle drifts in VFS-layer locking protocols over a decade of evolution [20].

D. Repair Cost Analysis

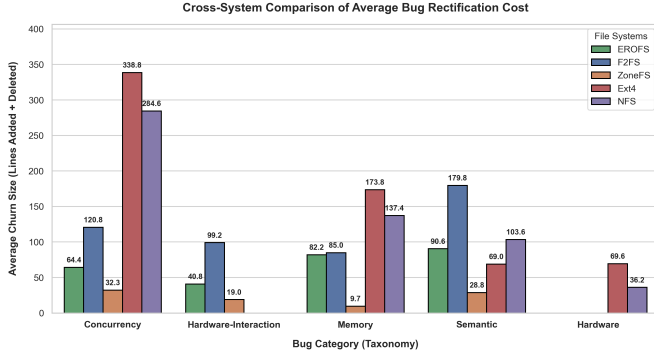


Fig. 19: Comparison of Average Repair Cost for the Top 5 Significant Bugs per Category (Since 2016).

1) *Macroscopic Overview*: To prevent the statistical dilution of core architectural defects by a vast number of trivial, lightweight fixes, this study employs a “Pareto-focused” auditing strategy. We conducted in-depth audits and calculated the average statistics for the top 5 non-trivial defects (ranked by code churn) within each category. This approach amplifies and precisely captures the maximum refactoring overhead and fundamental design pain points encountered by each system when resolving conflicts in core semantics, concurrency, or hardware interfaces.

As illustrated in Fig. 19, the average repair churn for **F2FS** in the *Concurrency* and *Semantic* categories is remarkably high. This quantitative evidence suggests that conflicts within F2FS’s core Inode locking mechanisms, multi-threaded I/O paths, and Garbage Collection (GC) pipelines are exceptionally intricate. Resolving such issues often necessitates a structural overhaul of the system’s state machine rather than localized patches.

As a read-only filesystem, **EROFS** exhibits relatively low costs for concurrency and hardware interaction. However, its architectural complexity is concentrated in high-performance in-memory decompression, resulting in noticeably higher repair costs for *Memory Management* and *Semantic* logic [14].

In contrast, **ZoneFS** demonstrates a high degree of defect locality due to its minimalist passthrough design. Even for the most complex *Hardware-Interaction* defects, the average repair requires only 19.0 lines of code (e.g., implementing a dual-circuit-breaker branch). This significantly lower churn showcases the inherent maintenance advantages of lightweight, hardware-aligned architectures.

**Note: Due to the minimal codebase and low total bug count of ZoneFS, while 5 samples were audited for Semantic bugs, only 3 significant samples were available for other categories. Nevertheless, the averaged statistics remain representative of the localized micro-adjustment trends within its evolution history.*

2) *Qualitative Attribution of High-Churn Defects*: To delve into the structural vulnerabilities of the investigated filesystems, we categorize the root causes of high-repair-cost (high-churn) defects into three fundamental patterns.

A. Cognitive Mismatch and Boundary Violations in Cross-Subsystem Interface Contracts

Modern kernels utilize generalized virtual infrastructures to decouple storage streams. While providing abstraction convenience, this design conceals potential conflicts in implicit cross-system assumptions. When a local filesystem implementation misaligns with the rigid contracts, lifecycles, or evolutionary traits of high-level frameworks, defects often cascade across modules, necessitating expensive refactoring. Under this attribution, five representative cases expose such boundary violations:

- **ext4** (4ba3fcdd7e): Lack of protection for cross-layer callbacks during concurrent unmounting triggered a *Use-After-Free* (UAF) vulnerability. Due to the impossibility of a direct fix within the VFS layer, developers were forced to deprecate generic callbacks and introduce a standalone *Shrinker*, causing an architectural shift in the memory reclamation path.
- **ZoneFS** (c1c1204c0d0): Returning zero length during out-of-bounds reads directly violated the *iomap* framework’s interface contract. This semantic deviation bypassed framework assertions and trapped the kernel iterator in an infinite loop.
- **nfs** (e3b2cf6e5db): Directly using raw network namespace pointers as hash and comparison keys leaked kernel pointers and bypassed *KASLR* defenses. Fixing this necessitated a reconstruction of the security isolation boundaries within the kernel control plane.
- **EROFS** (3acea5fc335): Adherence to a hardcoded global *PAGE_SIZE* assumption failed during sub-page block adaptation, triggering a cross-layer refactoring involving 14 source files.
- **EROFS** (5b9654efb60): Internal iterators lacked cross-page awareness for multi-page *Folio* carriers, rendering traditional single-page scanning mechanisms obsolete and forcing an upgrade to a multi-level iterator architecture.

Core Systematic Problem: Filesystem implementations frequently rely on localized physical assumptions or static contracts. When high-level frameworks evolve or enter extreme edge states, the lack of dynamic awareness regarding cross-system implicit contracts causes localized boundary failures to propagate into global architectural tremors.

B. Semantic and State Conflicts Induced by Horizontal Interweaving of Multidimensional Features

As filesystems evolve toward complex feature matrices (e.g., high concurrency, compression, and *bigalloc* space management), internal logic transforms into an intricate web of interdependencies. High-cost defects often erupt where multiple advanced features intersect at a single physical or logical point. Without a global view to govern concurrent characteristics,

TABLE III: Qualitative Analysis of Root Causes and Architectural Impacts for High-Cost Defects

Commit Hash	FS	Type	Churn	Files	Core Inducement	Architectural Cascade Effect
50fa53ec	F2FS	Sync	187	6	Missing async barriers causing disordered physical I/O submission.	Breakage of critical node chains, compromising power-cut consistency.
787caf1b	F2FS	Sem.	331	2	Incorrect priority of inline optimization over compression features.	Necessitated global decision path and feature matrix refactoring.
3acea5fc	EROFS	Sem.	265	14	Hardcoded PAGE_SIZE assumptions in physical block mapping.	Failure in sub-page block adaptation, causing cross-layer refactoring.
5b9654ef	EROFS	Mem.	167	1	Iterator lacks awareness of multi-page Folio boundaries.	Forced the upgrade of single-page scanners to multi-level iterators.
ebfd68cd	ZoneFS	Sync	78	1	Async I/O dependency on stale in-memory <i>i_size</i> .	Bypassed general validation logic, leading to address logic cohesion issues.
c1c1204c	ZoneFS	Sem.	94	1	Out-of-bounds read returning zero length, violating iomap contract.	Triggered framework assertion failures and infinite kernel loops.
9fe67149	Ext4	Sync	372	5	Improper accounting of “implicit sharing” in bigalloc delayed allocation.	Disrupted the VFS-allocator commitment chain, causing system-wide failures.
4ba3fcdd	Ext4	Sync	369	5	Cross-layer callback causing Use-After-Free (UAF) during concurrent unmount.	Forced the deprecation of VFS callbacks in favor of a standalone log-layer shrinker.
5b2095d0	NFS	Sync	456	1	Missing enum definitions and parameter type mismatch in Tracepoints.	Compromised observability data fidelity, impacting distributed telemetry chains.
e3b2cf6e	NFS	Mem.	323	15	Direct use of raw namespace pointers as hash keys (KASLR bypass).	Reconstructed the security isolation boundary of the kernel control plane.

*Note: Commit hashes are truncated to the first 8 characters. “Sem.” stands for Semantic and “Sync” for Concurrency/Synchronization.

local control flows easily produce semantic conflicts with the implicit states of other features, as evidenced by:

- **F2FS** (787caf1bdcd): Incorrectly prioritizing *inline optimization* over *compression* led to a semantic priority error that necessitated a total reconstruction of the global decision paths and the feature matrix.
- **ext4** (9fe671496b6): Deleting allocated extents under the *bigalloc* mechanism failed to identify *delayed allocation* reservations sharing the same physical cluster. This accounting inconsistency disrupted the commitment chain between the VFS layer and the block allocator, leading to system-wide failures.

Core Systematic Problem: When a system enables multiple optimization features simultaneously without robust feature isolation, the divergence between physical media states and in-memory logical states causes semantic fragmentation, resulting in system-level failures or forced architectural reorganization.

C. Vision Disjunction Resulting from Asynchronous Control Failures between Physical Media and the Control Plane

Filesystems must maintain media consistency and telemetry observability while pursuing throughput. Empirical data suggests that high-cost defects often stem from the failure to establish effective atomic barriers or synchronization protocols during asynchronous control flows, leading to a rupture in the state vision.

- **F2FS** (50fa53eccf9): The absence of necessary asynchronous barriers caused disordered physical I/O submissions. This led to broken critical node chains during crash recovery, compromising the filesystem’s power-cut consistency guarantees.

- **ZoneFS** (ebfd68cd0c1): Asynchronous I/O paths relied on a stale in-memory view of the file size (*i_size*), bypassing general validation. Fixing this temporal vulnerability required a low-level reconstruction of the cohesive physical addressing logic.
- **nfs** (5b2095d0cef): Missing enum definitions and parameter type mismatches within tracepoints caused symbolic resolution failures in the asynchronous observability plane, compromising telemetry fidelity and triggering cascaded impacts across distributed operations.

Core Systematic Problem: When asynchronous designs are adopted for performance without commensurate logical barriers, microscopic temporal lags are magnified. This asymmetry between the memory vision and the actual physical entity (or distributed monitoring view) directly compromises consistency and operational security.

3) *Engineering Principles for Low-Level System Development:* Based on the analysis of high-cost defects, we propose the following engineering guidelines for developing low-level software, particularly for high-concurrency filesystems and kernel driver modules:

I. Defensive Contract Guidelines for Cross-Subsystem Interaction

Low-level modules are deeply embedded within core frameworks such as the VFS and memory management. Developers must interact with these subsystems from a defensive perspective to ensure interface robustness.

- *Explicit Handling of Boundary Contracts:* Coding solely for the “ideal flow” is strictly prohibited. Complete logical branches must be established for all potential boundary return values of general-layer interfaces, includ-

ing zero-length returns, retry requests, and specific error codes.

- *Cross-Layer Object Lifecycle Alignment*: When introducing cross-subsystem callback mechanisms, the target object’s lifespan must be rigorously evaluated under concurrent unmounting or extreme memory pressure. It is forbidden to assume that external frameworks will persistently hold internal resource locks for the subsystem.
- *Control Plane Security Isolation*: When passing pointers across privilege boundaries (e.g., user-to-kernel or between network namespaces), data sanitization or hash transformation must be implemented at the interface. Direct exposure of or reliance on sensitive identifiers like physical addresses is prohibited.

II. Abstract Decoupling Guidelines for Carrier Evolution

The evolution of storage media and kernel memory management carriers (e.g., the transition to Folios) is a long-term trend. Developers should use abstraction to shield against underlying physical differences.

- *Elimination of Hardcoded Physical Parameters*: Constants such as physical page size must not be hardcoded in scanning, traversal, or data processing logic. Instead, developers should invoke abstract offset functions and generic iterator interfaces provided by the kernel framework.
- *Carrier-Boundary-Aware Design*: Low-level iteration logic must be “container-aware.” The system should automatically recognize whether a processing object spans physical carrier boundaries (e.g., moving from a single page to a multi-page *Folio* cluster) and implement corresponding boundary overflow protections.

III. State Transactional Guidelines for Multidimensional Feature Coordination

As functional features stack, the correctness of a single logical path no longer guarantees global consistency. A global perspective must be established to manage feature interactions.

- *Explicitation of Feature Priority Matrices*: When multiple advanced features (e.g., inline optimization, transparent compression, delayed allocation) coexist, clear execution priorities and mutual exclusion rules must be defined via a state matrix to prevent semantic conflicts in complex decision paths.
- *Transactional Maintenance of State Ledgers*: Complex operations involving physical block release and logical reservation sharing must be treated as atomic transactions. Defensive state flags should be introduced during the intermediate synchronization window to block any unintended cut-ins from concurrent streams.

IV. Temporal Barrier Guidelines for Asynchronous Control Flows

High-performance systems leverage asynchrony to increase throughput, but strict barrier protocols must be established to prevent “vision disjunction.”

- *Mandatory Logical Barrier Protocols*: After performing critical metadata updates or synchronous writes, asynchronous writeback waits or physical barrier protocols must be explicitly triggered. Relying on the spontaneous flush mechanisms of underlying hardware is prohibited.
- *Atomized Re-validation of Asynchronous Paths*: Before performing final physical addressing, asynchronous I/O control flows must execute an atomized re-validation of critical state views (e.g., file size and offsets). Blindly trusting memory snapshots that may have been invalidated by concurrent races is prohibited.

V. High-Reliability Maintenance Guidelines for the Observability Plane

Diagnostic and tracing code should be treated as core components of the production environment, with maintenance standards equal to those of data-plane code.

- *Alignment of Type and Semantic Consistency*: Parameter passing for tracepoints and monitoring probes must strictly follow the same type-checking and enum mapping specifications as the production code to ensure telemetry fidelity.
- *Decoupling of Diagnostic Logic from Core Paths*: Monitoring logic involving deep symbolic resolution or complex computation should not be embedded within critical concurrent paths. Diagnostic logic must remain simple and self-contained to ensure that its own failure does not trigger systemic cascades through the operational monitoring chain.

CONCLUSION

This paper presents a ten-year empirical study analyzing 3,906 bug-fixing commits across five representative Linux file systems from January 2016 to April 2026. Building upon the foundational evolution analysis of earlier Linux storage stacks [1], our findings demonstrate that a file system’s defect profile is fundamentally dictated by its underlying hardware assumptions, concurrency models, and state-tracking complexities. While mature systems grapple with object lifecycle management within legacy subsystems, emerging hardware-specialized or distributed systems incur severe maintenance overheads driven by intricate asynchronous execution paths.

The empirical insights distilled from this analysis offer critical architectural guidance for next-generation storage system design:

- **Trade-offs Between Architectural Minimalism and Subsystem Compensation**: The contrast between the heavy compensation layer in F2FS [21] and the minimalist, passthrough philosophy of ZoneFS [22] underscores a fundamental trade-off. Integrating device-specific logic into core file system state machines severely escalates technical debt. Future architectures tailored for emerging hardware are poised to benefit from decoupled, lean abstractions that isolate physical management from virtual file semantics.
- **Transactional Governance for Feature Matrices**: As modern storage solutions trend toward multi-dimensional

feature co-existence, ad-hoc synchronization becomes unsustainable. Next-generation systems are encouraged to govern feature interactions through explicit priority matrices and strict transactional boundaries to prevent accounting drift between physical media and memory views.

- **Defensive Contract Enforcement and Telemetry:** Cross-subsystem contract mismatches with evolving virtual layers remain a primary source of structural regressions. It is therefore paramount for future file systems to implement rigid defensive programming boundaries, elevating interface contract validation and high-fidelity asynchronous telemetry to first-class architectural components.

REFERENCES

- [1] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *ACM Transactions on Storage (TOS)*, vol. 10, pp. 1 – 32, 2013. [Online]. Available: <https://api.semanticscholar.org/CorpusID:6594561>
- [2] J. Śliwinski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1 – 5, 2005. [Online]. Available: <https://api.semanticscholar.org/CorpusID:5832984>
- [3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. ICSE*, 2005.
- [4] M. Bjørling *et al.*, "Zns: Avoiding the block interface tax for flash-based ssds," in *Proc. USENIX FAST*, 2020.
- [5] X. Gao *et al.*, "Erofs: A compression-friendly read-only file system," in *Proc. USENIX ATC*, 2019.
- [6] V. Prabhakaran *et al.*, "Iron file systems," in *Proc. SOSP*, 2005.
- [7] C. Lee *et al.*, "F2fs: A new file system for flash storage," in *Proc. FAST*, 2013.
- [8] L. Lu *et al.*, "A study of linux file system evolution," in *Proc. FAST*, 2013.
- [9] H. S. Gunawi *et al.*, "Understanding and finding system-specific error handling bugs," in *Proc. ASPLOS*, 2008.
- [10] S. Shepler, M. Eisler, and D. Noveck, "Network file system (nfs) version 4 minor version 1 protocol," Tech. Rep., 2010.
- [11] M. Bjørling *et al.*, "Zns: Avoiding the block interface tax for flash-based ssds," in *Proc. USENIX FAST*, 2020.
- [12] J. Śliwinski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. MSR*, 2005.
- [13] A. Chou *et al.*, "An empirical study of operating system errors," in *Proc. SOSP*, 2001.
- [14] X. Gao *et al.*, "Erofs: A compression-friendly read-only file system," in *Proc. USENIX ATC*, 2019.
- [15] N. Palix *et al.*, "Faults in linux: Ten years later," in *Proc. ASPLOS*, 2011, pp. 305–318.
- [16] Z. Yin *et al.*, "An empirical study on real-world concurrency bugs in open source software," in *Proc. SOSP*, 2011, pp. 327–342.
- [17] C. Min *et al.*, "Understanding the concurrency bugs in the linux kernel," in *Proc. ASPLOS*, 2021, pp. 706–719.
- [18] T. J. Ostrand *et al.*, "The distribution of faults in a large software system: Forecasting the locations of defects," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 67–82, 2005.
- [19] S. Kim and E. J. Whitehead, "How long do bugs live?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, 2006, pp. 121–124.
- [20] L. Tan *et al.*, "icheck: A static analysis tool for inferring and checking file system protocols," in *Proc. SOSP*, 2015.
- [21] C. Y. Lee, D.-Y. Sim, J. Y. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *USENIX Conference on File and Storage Technologies*, 2015. [Online]. Available: <https://api.semanticscholar.org/CorpusID:14566236>
- [22] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis, "Zns: Avoiding the block interface tax for flash-based ssds," in *USENIX Annual Technical Conference*, 2021. [Online]. Available: <https://api.semanticscholar.org/CorpusID:235818273>