ConFS: A Container-Aware Userspace File System with Enhanced Isolation

Zirui Wang*†, Ying Wang*, Dejun Jiang*†‡
*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences;
†University of Chinese Academy of Sciences; ‡Zhongguancun Laboratory
{wangzirui22z, wangying2023, jiangdejun}@ict.ac.cn

***Abstract*—Containers provide applications with isolated execution environments and regulated resource usage, but file system operations from different containers still execute over shared kernel state and code paths. As a result, file system isolation is weakened in three aspects: shared metadata can undermine access isolation, shared in-kernel components can propagate faults across co-located containers, and shared in-kernel resource management can cause cross-container interference.**

In this paper, we present ConFS, a container-aware userspace file system that reorganizes file system isolation around container domains. ConFS introduces container domains as the isolation unit and maintains separate file system state for each domain. For access isolation, ConFS confines each operation to domain-local state, eliminating the shared metadata paths exploited by container-escape attacks. To isolate file system faults, ConFS moves execution to userspace and maintains per-domain state; fast recovery then restores file system service within tens of milliseconds after a crash. For resource isolation, per-domain metadata, buffering, and I/O control prevent cross-domain interference. Our evaluation shows that ConFS provides enhanced isolation with less than 2% performance degradation compared to existing userspace file systems.

***Index Terms*—Userspace file system, Container, Isolation**

I. Introduction

Containers (e.g., Docker [1] and Podman [2]) have become the standard deployment unit in cloud platforms, offering elastic and lightweight virtualization that enables multi-tenant workloads to share the host operating system [3]–[5]. To isolate co-located containers, each container is given an independent execution environment, with resource usage regulated through kernel mechanisms such as namespaces [6] and cgroups [7].

The container file system is a critical component through which containers interact with the host for data persistence and sharing. Each container has an isolated file system view created through the mechanisms such as mount namespaces [8], union file systems [9], [10], and bind mounts [11]. However, file system operations from co-located containers are still processed by the shared host kernel file system. This creates an isolation boundary mismatch: containers are intended to serve as the isolation unit, but the kernel file system is shared across containers and not aware of container boundaries. This mismatch subsequently leads to weak isolation in three aspects.

First, for access isolation, although mount namespaces [8] restrict each container's file system view, file system operations are still executed through the shared *virtual file system* (VFS) using host file system metadata. This shared execution path exposes attack surfaces that have led to over 40 container-escape vulnerabilities [12]–[14]. Second, for fault isolation, file system operations from all containers are processed through shared in-kernel file system components and state, such as the VFS, page cache, and file system metadata, within the same kernel address space, causing a fault triggered by one container to disrupt all co-located containers [15], [16]. Third, for resource isolation, containers rely on globally managed file system metadata and data handling in the host kernel, so heavy metadata operations or I/O activity in one container can degrade the performance of others.

Existing approaches [12], [13], [17], [18] mainly address these issues in a reactive manner, by patching individual vulnerabilities in the kernel file system or adding runtime checks along the shared kernel path. Although being effective for known problems, these approaches do not eliminate the root cause of weak isolation: containers still rely on the same host kernel file system and shared state, so similar issues continue to emerge over time [12], [13]. We argue that achieving stronger container file system isolation requires rethinking the file system boundary itself. Rather than inheriting the host-wide boundary, the file system should isolate unrelated containers while preserving data sharing among containers.

In this paper, we present ConFS, a container-aware userspace file system that refines the file system boundary around container domains, which group containers according to their data-sharing relationships. By aligning file system execution and state management with this domain boundary, and enabling fast recovery through domain-partitioned state reconstruction, ConFS provides a unified foundation for access, fault, and resource isolation. Under this design, each file system operation is processed using its own domain's file system state, eliminating the shared metadata paths exploited by container-escape vulnerabilities. Failures are confined to individual domains, and fast recovery reconstructs the file system service and the runtime state needed by blocked applications after a crash. Finally, by separating metadata and data management across domains and enforcing per-domain I/O control, ConFS prevents cross-domain resource interference caused by globally shared file system management.

We evaluate ConFS using micro-benchmarks, real-world applications, and CVE analysis. For access isolation, we analyze container-escape CVEs from 2017 to 2025 and verify

that ConFS eliminates the shared metadata exploited by these vulnerabilities. For fault isolation, ConFS restores file system service for containerized applications within tens of milliseconds. For resource isolation, ConFS prevents cross-container interference in both metadata throughput and I/O bandwidth. Overall, ConFS incurs negligible performance overhead (less than 2% in most cases) compared to a userspace file system without isolation support.

In summary, we make the following contributions:

- We identify the isolation boundary mismatch between the intended container-level isolation unit and the shared host kernel file system, causing container-escape vulnerabilities, host-wide fault propagation, and cross-container resource interference.
- We propose ConFS, a container-aware userspace file system that introduces container domains as the isolation unit and aligns file system execution and state management with domain boundaries, while enabling fast recovery through domain-partitioned state reconstruction.
- We conduct extensive experiments to evaluate ConFS with microbenchmarks and real-world applications, showing enhanced isolation with less than 2% performance overhead.

II. Background and Motivation

A. Container Storage

Containers are a lightweight OS-level virtualization technology that allows multiple applications to share a common host operating system while providing isolation at the process, memory, and other system resources [6], [19]. In practice, containerized applications access data through multiple mechanisms, including image-based file systems (e.g., OverlayFS [9], [10]), host-managed volumes, and direct access to underlying storage devices. Despite these differences, all file accesses ultimately traverse the same host kernel file system stack. File operations issued via system calls are processed by the *virtual file system* (VFS) and dispatched to underlying file systems (e.g., Ext4 [20] or XFS [21]) on physical storage devices. As a result, storage operations from different containers are handled through shared in-kernel file system components and state that are not organized around container-level isolation boundaries [19], [22].

To provide isolation between co-located containers, existing container runtimes rely on *mount namespaces* [6] to separate file system views, giving each container a logically isolated directory hierarchy. In addition, *cgroups* [7] are used to regulate resource usage, such as limiting I/O bandwidth through the `io` controller.

However, these mechanisms primarily enforce isolation at the level of namespace visibility and resource accounting, while file operations from co-located containers are still handled through the shared host kernel file system. This leads to container-escape vulnerabilities, host-wide fault propagation, and cross-container resource interference. We next examine these limitations in detail.

```
void ext4_xattr_set_entry() {
  disk_size = ReadXattrSizeFromDisk(); // read a
      size value controlled by the request
  write_len = AlignXattrSize(disk_size); // compute
      how many bytes to clear
  xattr_buf = GetAllocatedXattrBuffer(); // buffer
      has only the expected fixed size
  memset(xattr_buf, 0, write_len); // if write_len
      is too large, the write goes out of bounds
}
```

Listing 1: Simplified pseudocode of CVE-2019-19319

B. Weak Access Isolation

Access isolation requires that a container can only access files within its own file system, unless explicitly shared. Currently, this isolation is primarily enforced by *mount namespaces*, which restrict the visible directory tree of a container to a designated subdirectory of the host file system (e.g., `/var/lib/docker/.../rootfs`).

However, this mechanism only restricts which paths are visible to a container, without isolating how those paths are resolved. All file accesses are still resolved through the shared VFS and global file system metadata, making access isolation depend on shared metadata that is not isolated across containers. For example, the path-resolution process involves multiple steps (e.g., dentry traversal, symbolic link resolution, mount lookup) that operate over global, mutable metadata structures [23]. As a result, a path initially resolved within a container's scope may later be resolved under different metadata state, violating the intended access boundary. This issue is exemplified by path re-resolution vulnerabilities such as CVE-2018-15664 [24], where a TOCTTOU race allows attackers to redirect file accesses outside the container's root directory. More broadly, over 40 vulnerabilities have been reported in container file systems due to inconsistencies in path resolution semantics [12]–[14].

Importantly, these vulnerabilities are not merely implementation bugs, but stem from the design that containers share file system metadata and resolution paths in the host kernel. Existing mitigations, such as Patrol [12] and PACED [13], introduce additional runtime checks to validate resolved paths. However, they still operate over shared file system metadata, and therefore cannot eliminate the root cause of weak access isolation.

C. Weak Fault Isolation

Fault isolation requires that each container form an independent failure domain. However, in current systems, file system operations from all containers are handled through shared in-kernel file system components within the same kernel address space. As a result, a fault triggered by one container can lead to inconsistent metadata that affects other containers, or in more severe cases, crash the entire host [15], [16].

We illustrate this risk using a real-world Ext4 vulnerability, CVE-2019-19319 [25]. As shown in Listing 1, a crafted `setxattr` operation can cause an out-of-bounds memory

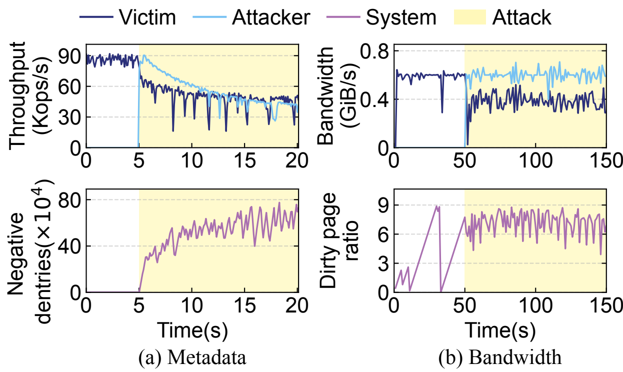

Fig. 1: Resource interference test on Ext4.

write in the Ext4 implementation, leading to a kernel crash. We reproduce this vulnerability in a controlled setup by reintroducing the vulnerable logic into the host Ext4 file system. When a container issues the crafted request, the shared kernel file system code is triggered, causing both the container and the host to crash simultaneously.

Importantly, this vulnerability is not specific to Ext4 alone, but reflects a broader isolation problem that containers still rely on shared in-kernel file system components and state, without an independent failure domain. Existing approaches attempt to mitigate such failures through recovery mechanisms. For example, Membrane [18] enables recovery via checkpointing and in-memory state tracking, while Ananke [15] replays logged requests in a user-space recovery service. However, these approaches still operate over shared file system state and cannot prevent faults from propagating across containers.

D. *Weak Resource Isolation*

Resource isolation is weakened because file system resources in the kernel are managed through globally shared structures and mechanisms. Existing resource isolation primarily relies on *cgroups* to regulate resource usage at the device level, such as limiting bandwidth or I/O rates among containers [7], [26]. However, cgroups do not isolate shared file system resources above the device layer. In current container stacks, such shared resources arise mainly in two forms: metadata caches and data buffering and writeback management. Consequently, containers can still suffer from unintended interference even when explicit device-level resource limits are enforced. We next examine two representative cases of weak resource isolation.

Interference on shared metadata management. File system metadata caches in the VFS, such as dentries and inodes, are managed in globally shared data structures and are not isolated among containers [17]. This lack of isolation causes cross-container interference. For example, when a process opens a non-existent file, the VFS creates a negative dentry and inserts it into the global hash table. An attacker container can generate a large number of negative dentries that increase hash collisions, thereby increasing the lookup overhead in the global hash table and degrading metadata access performance of other containers.

We demonstrate this effect using an experiment on Ext4. A victim container runs a metadata-intensive workload, while an attacker container continuously issues accesses to non-existent files to generate negative dentries. Both containers operate within their assigned resource limits, and the experiment does not saturate overall system resources or device bandwidth. As shown in Figure 1(a), the victim sustains a stable throughput of about 87 Kops/s during the first 5 seconds. At the fifth second, the attacker begins generating negative dentries. The dentry lookup overhead increases with the increased negative dentries. Correspondingly, the victim's throughput immediately drops by about 25% and eventually falls to roughly 47 Kops/s. Prior work [17] attempts to mitigate such interference through reactive throttling, but file system metadata remains shared across containers, so contention on shared metadata structures can still degrade performance.

Interference from global I/O management. In current container stacks, data I/O for co-located containers is still managed globally by the kernel. In particular, buffering and writeback remain globally coordinated across containers. As a result, one container's write activity can interfere with the normal data accesses of others. One example is the kernel's global dirty-page management, where writeback is triggered for all containers once the global dirty-page ratio exceeds the background writeback threshold. On the Ubuntu server we use, the default value of this threshold (`dirty_background_ratio`) is 5%.

We demonstrate this interference using an experiment in which a victim container is assigned with a fixed bandwidth of 0.6 GiB/s via *cgroups*, and an attacker container continuously generates buffered writes. The experiment is configured so that the observed slowdown is not due to the victim exceeding its configured bandwidth limit or to simple saturation of overall system resources. As shown in Figure 1(b), the victim's bandwidth drops noticeably from 0.6 GiB/s to 0.4 GiB/s after the 50th second, due to the increased dirty page ratios and global writeback interference. This slowdown arises because data buffering and writeback are globally managed across containers, so one container's write activity can interfere with another's normal data access even under explicit device-level limits.

E. *Opportunity and Goals*

All these limitations stem from the fact that co-located containers still rely on the same host kernel file system handling and shared file system state. As a result, the isolation boundary exposed to containers does not align with the internal state boundaries of the file system. This observation suggests that achieving enhanced isolation requires rethinking how file system execution and state are organized for unrelated containers.

One promising direction is to move file system logic out of the shared kernel and explicitly organize file system state around container isolation boundaries. By separating

file system state, including metadata and data management, for unrelated containers, such a design can eliminate the root cause of cross-container interference. Motivated by this insight, we redesign the file system architecture as a container-aware userspace file system with enhanced isolation, avoiding shared file system state where sharing is not intended.

To avoid ambiguity, we clarify the scope of our design. ConFS does not aim to eliminate file system bugs or guarantee the correctness of file system implementations. Instead, it aims to isolate the effects of such bugs and confine their impact within intended isolation boundaries. Specifically, ConFS ensures that file system state is not shared across unrelated containers, so that faults, mis-resolutions, or resource contention in one container do not propagate to others. In this sense, our goal is to provide enhanced isolation and containment, rather than to improve the correctness of individual file system operations. In other words, we do not eliminate bugs; we eliminate their cross-container impact.

III. DESIGN OF CONFS

A. Architectural Overview

To address the boundary mismatch between container isolation and the shared host kernel file system, we propose ConFS, a container-aware userspace file system that aligns file system execution and state management with container domains, which serve as the unified boundary for access, fault, and resource isolation.

A straightforward approach is to assign each container an independent file system instance to fully isolate file system state. However, this approach is impractical in containerized environments, where multiple containers share the same storage device. Independent file system instances would lack a unified view of device-level resource management, including space allocation, I/O scheduling, and buffering, leading to inefficient utilization and uncoordinated access.

Moreover, the container itself is often not the right unit of storage isolation. In practice, a single application may consist of multiple containers that cooperate and share data for coordination (e.g., within a pod [27]). Therefore, isolation should be defined at the granularity of groups of containers that intentionally share data, rather than at the granularity of individual containers. To capture this requirement, ConFS introduces container domains as the isolation unit. A container domain groups containers that intentionally share data, while isolating them from unrelated containers. Within a domain, containers may still have different file system views and access permissions, enforced by existing mechanisms such as namespaces. The domain abstraction defines the scope of shared file system state, including metadata, data buffers, and associated runtime state, but not the visibility of individual files to applications.

To enforce both sharing and domain-level isolation, ConFS uses a single file system process to manage the storage device while maintaining independent file system state for each domain. To realize this design, ConFS adopts a split architecture, consisting of a per-container userspace library (LibFS) and a host-resident file system server (FSServer), as shown in Figure 2. LibFS binds each request to its container domain, while FSServer multiplexes the shared device and maintains separate file system state for each domain. This architecture enables efficient device sharing while preventing cross-domain exposure of file system state. More importantly, it aligns file system execution and state management with the domain boundary, so that operations are attributed, executed, recovered, and regulated within the same isolation scope.

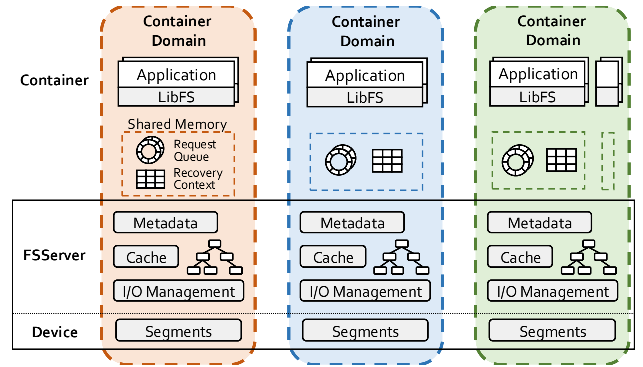

Fig. 2: ConFS architecture.

Per-container userspace library (LibFS). LibFS is a shim library that is dynamically linked to the application at launch, intercepting file-related system calls without requiring application recompilation. The intercepted calls are translated into file system requests and forwarded to FSServer via shared-memory queues bound to container domains. Requests are enqueued by LibFS and processed by FSServer in a polling manner. These queues not only deliver requests efficiently but also attribute each operation to its originating domain through the communication path, enabling reliable domain attribution and forming the basis for subsequent recovery.

Host-resident userspace file system server (FSServer). FSServer runs as a userspace daemon on the host and serves all container domains over a shared storage device. It leverages the SPDK userspace I/O stack [28] for direct device access while safely multiplexing the device across domains.

FSServer maintains separate file system state for each container domain, including metadata, data buffers, and runtime state. Requests from all domains are processed concurrently by a pool of worker threads. Each worker determines the target domain based on the queue on which a request arrives and executes the operation within that domain's file system state. In this way, ConFS separates file system state at the domain level while still sharing device management across domains.

To support safe device sharing, FSServer maintains per-domain address mappings and ensures that all I/O remains within the storage range allocated to the target domain. This design centralizes device management in a single process while preventing cross-domain exposure of storage space.

B. Access Isolation

To isolate storage without breaking sharing semantics in containerized environments, ConFS uses container domains as the unit of access isolation. In contrast to conventional container file systems, which resolve operations over shared file system state, ConFS confines each operation to domain-local state.

ConFS enforces access isolation by ensuring that both the identity of a request and its resolution are confined to the same domain-bound state. This is achieved through two invariants: (1) each request must be reliably attributed to its originating container domain, and (2) each request must be resolved only within that domain's file system state. These invariants are enforced through trusted domain attribution and domain-local file system execution, respectively.

Trusted domain attribution. The first step of access isolation is to ensure that FSServer can trust the domain identity of each incoming request. ConFS achieves this by binding each request queue to a single container domain and deriving the request's domain solely from the queue on which it arrives, rather than from user-provided request fields. As a result, domain identity is enforced by the communication channel itself and cannot be spoofed or tampered with by applications.

To establish this binding, ConFS sets up a trusted control channel between each container and FSServer at initialization. FSServer authenticates the connecting processes using kernel-provided credentials (e.g., via Unix domain sockets) and creates shared-memory request queues that are permanently bound to the container's domain. All subsequent requests are routed exclusively through these domain-bound queues.

Domain-local File System Execution. After identifying the target domain, FSServer must also ensure that each operation is resolved entirely within that domain's file system state. ConFS achieves this by partitioning both on-disk and in-memory file system state at the domain level.

Each domain maintains independent file system state that is not shared across domains. As a result, file operations such as pathname resolution, metadata lookup, and cache access are performed using domain-local state rather than globally shared state. Within a domain, containers may still share files or volumes that are intentionally assigned to that domain. Across domains, operations are resolved against separate file system state, preserving isolation without requiring per-container duplication.

C. Fault Isolation

In multi-tenant container environments, file system faults can propagate across containers because file system operations from all containers are handled through shared in-kernel file system components within the same kernel address space. As a result, a fault triggered by one container, such as an out-of-bounds write during metadata handling, can corrupt shared file system state or crash the kernel, thereby disrupting all co-located containers.

To provide fault isolation, ConFS first moves file system execution out of the shared kernel context, preventing file system faults from crashing the host or directly terminating application processes. It further partitions file system state by domain, ensuring that each domain maintains its own metadata and runtime state and that all operations are resolved against domain-local state rather than shared state across domains.

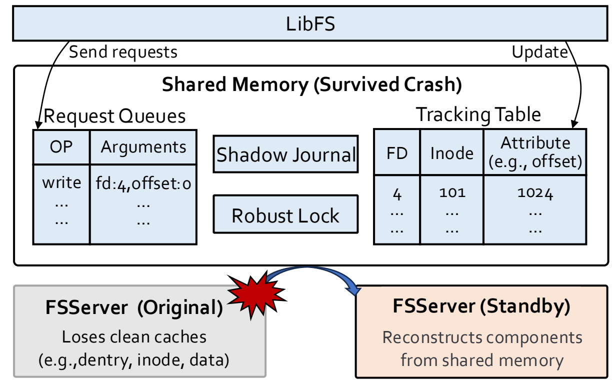

Fig. 3: Data Structures for Fast Recovery in ConFS.

A fail-stop crash of the FSServer, however, can interrupt file system service for all domains, because ConFS uses a single FSServer process. We therefore adopt a crash model in which only the FSServer may fail, while the host system and application processes survive. Under this model, only the FSServer's private in-memory state is lost; on-disk state and the shared-memory recovery state (between FSServer and LibFS) remain available. By contrast, failures of LibFS or application processes are container-level faults and lie outside the scope of ConFS, as they do not corrupt the FSServer's file system state. Host-system crashes are also outside our main fault model and are handled by conventional on-device journal replay.

To handle a fail-stop crash of FSServer, ConFS provides fast recovery that restores file system service with minimal disruption to running applications. Such recovery must satisfy three requirements: (1) quickly resume file system service, (2) restore crash-consistent file system state, and (3) reconstruct the runtime state needed by applications that remain alive across the crash, including open file descriptors, file offsets, and in-flight updates. ConFS addresses these requirements through three mechanisms: **R1 (Fast Takeover)**, which resumes service with minimal interruption; **R2 (Consistency Restoration)**, which restores crash-consistent state; and **R3 (Runtime Reconstruction)**, which rebuilds application-visible runtime state. After takeover, R2 and R3 operate independently on each domain's state and can proceed in parallel across domains, so recovery does not mix or propagate state across domain boundaries. Figure 3 shows the data structures that support recovery in each mechanism.

R1: Fast Takeover. Fast takeover minimizes the interruption between an FSServer crash and the resumption of file system service. A normal restart would require process creation and initialization, including storage-driver setup and runtime resource allocation. Instead of performing these steps

on the recovery path, ConFS prepares a standby process in advance and switches to it immediately after a crash.

For failure detection, ConFS uses a kernel-supported robust lock shared between FSServer and the standby process. FSServer holds the lock during normal execution, while the standby process blocks on it (shown as the robust lock in Figure 3). When FSServer crashes or exits, the kernel releases the lock and wakes the standby process, enabling immediate takeover without external monitoring.

The standby process completes its startup and initialization before a crash occurs and remains in a ready state. After wakeup, it proceeds directly to consistency restoration and runtime reconstruction without re-executing the normal startup path. Thus, R1 removes fixed startup overhead from recovery and shortens the service interruption before state reconstruction begins.

R2: Consistency Restoration. After takeover, the standby process must restore a crash-consistent file system state before normal service can resume. ConFS adopts ordered journaling, similar to conventional file systems such as Ext4. Under this model, recovery needs to replay only committed metadata updates that have not yet reached their final on-device locations, while data blocks are guaranteed to be persisted before the corresponding metadata commit.

To reduce recovery latency and avoid blocking other domains, ConFS avoids bulk journal-block reads from the recovery path. Conventional journaling recovery requires reading journal blocks from the device, introducing additional I/O and prolonging recovery latency. Instead, ConFS maintains a *shadow journal* in shared memory that mirrors the on-device journal (as shown in Figure 3).

During normal operation, journal updates are first recorded in the shadow journal and then flushed to stable storage. Because the shadow journal resides in shared memory that remains accessible to the standby process, it is still available after an FSServer crash. The shadow journal does not change the durability boundary; it only changes the source from which committed journal records are replayed after a crash. During recovery, the standby process consults the on-device journal header to determine the committed transaction prefix and replays those transactions directly from the in-memory shadow journal. This avoids journal-block reads from the device while preserving the same commit semantics as conventional ordered journaling.

By performing replay entirely in memory and applying it only to the corresponding domain state, R2 shortens recovery time without mixing file system state across domains. In the rare case of a full host crash, where shared memory is lost, ConFS falls back to conventional on-device journal replay to preserve correctness.

R3: Runtime Reconstruction. At the time of a crash, file system operations may be in different stages: some have been fully persisted, some have partially executed but not yet been committed, and others have not executed at all. An FSServer crash loses its private in-memory state, including cached metadata and updates that have not yet reached persistent storage. In contrast, shared-memory state maintained by ConFS, such as tracking tables and retained requests (as illustrated in Figure 3), remains available to the standby process after takeover.

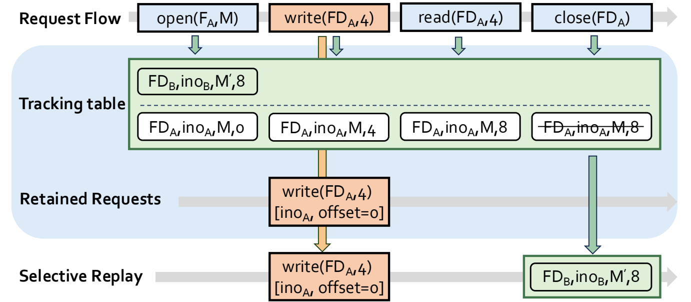

Fig. 4: Example operation sequence with tracking table and retained requests. F: file, M: open mode, ino: inode number, FD: file descriptor; subscripts A and B distinguish two files.

R3 bridges this gap in two steps. First, ConFS selectively replays operations whose effects have not yet reached persistent storage, thereby restoring lost metadata and data updates. Second, R3 reconstructs the runtime state needed by applications that remain alive across the crash. This step is necessary because POSIX file system operations are stateful and depend on runtime context such as file descriptors, access modes, and file offsets. For example, in a sequence such as `open` followed by `write`, the write operation depends on the file descriptor returned by the open call and on file offsets that evolve over time. Without reconstructing this runtime state, replay alone would be insufficient to correctly interpret subsequent operations.

Figure 4 illustrates how ConFS tracks and recovers both kinds of state.

(i) Reconstructing runtime state. R3 must recover two related kinds of state. First, it must restore the current per-process runtime state that applications continue to rely on after a crash, such as open file descriptors and their offsets. Second, replay in part (ii) requires request-specific state associated with retained operations, such as the inode referenced by a file descriptor and the corresponding file offset. ConFS preserves this second kind of state together with retained requests and uses it during replay. This subsection focuses on reconstructing the per-process runtime state visible to applications.

In conventional file systems, such runtime state is typically not recovered, because applications restart and rebuild it after a failure. In contrast, ConFS aims to preserve application-visible semantics across an FSServer crash, so it must explicitly restore the runtime state observed by still-running applications. A straightforward approach would be to re-execute prior operations to rebuild this state, but this would introduce unnecessary replay work and dependency tracking.

Instead, ConFS records runtime state explicitly in a per-process tracking table maintained by LibFS in shared memory, allowing it to survive an FSServer crash. Each entry records the state of an open file, including its file descriptor, inode

number, open mode, and current offset. The table is incrementally updated during normal execution: operations that change the set of open files (e.g., open, close) insert or remove entries, while operations that modify file position (e.g., read, write, lseek) update the stored offset.

During recovery, the standby process reads the tracking table and reconstructs the corresponding file descriptors and offsets directly, without replaying dependent operations to rebuild this context. Figure 4 illustrates this process. For example, `open`(F_A, M) creates an entry $\langle FD_A, ino_A, M, 0\rangle$, and subsequent operations update the offset accordingly. If a crash occurs before close, the standby process reconstructs the open file state directly from the table.

By restoring runtime state explicitly, ConFS avoids replaying the dependency chain of earlier operations while preserving correct application-visible semantics.

(ii) Replay of non-persistent updates. Reconstructing runtime state alone is insufficient. An FSServer crash may also lose file system updates whose effects have been applied in memory but have not yet reached persistent storage. For example, after `write`$(FD_A, 4)$ in Figure 4, the written data may still reside in in-memory buffers at the time of the crash. Restoring the file descriptor and offset does not recover this data. Therefore, R3 must replay such operations to restore the lost effects of non-persistent updates.

To support this, ConFS retains state-modifying requests in the shared-memory request queue and replays them during recovery. For each retained request, ConFS records the information needed for replay, including the resolved inode number, the file offset, and the associated data buffer. This recorded information also includes the request-specific state described in part (i), ensuring that replay proceeds correctly after recovery.

Replay restores the effects of operations directly, without re-executing the full file system path. For example, replaying a write operation reads data directly from the shared-memory buffer and writes it to the recorded inode and offset, bypassing path resolution and permission checks. This avoids unnecessary work and reduces replay latency.

However, replay is not uniform across all operations, because metadata and data updates have different persistence semantics at crash time. For metadata, replay is coordinated with journaling recovery. Because journaling enforces atomicity at transaction boundaries, metadata updates do not expose partial persistence at crash time. They are either already committed and recovered by R2, or not yet committed and therefore handled by R3. Data updates are different. Their effects may be only partially persisted when the crash occurs, so R3 replays the corresponding operations to restore the missing effects. In this way, R3 covers all non-persistent updates, including both data-modifying operations (e.g., `write`) and metadata-only operations (e.g., `create`, `unlink`). This distinction also means that replay must restore only missing effects rather than blindly reapplying all retained updates.

To avoid redundant replay, ConFS tracks the durability of each retained request and skips requests whose effects have already reached persistent storage. In particular, for data-modifying operations such as writes, ConFS records whether the corresponding data has already been persisted. During recovery, R3 replays only those updates whose effects are not yet reflected on persistent storage, thereby avoiding redundant writes and improving replay efficiency.

Crash Recovery and Application Resumption. To ensure correctness, recovery proceeds in three steps. First, the system replays non-persistent retained requests to restore missing file system updates. This step ensures that operations whose effects were not yet reflected on persistent storage are reapplied. This order is necessary because reconstructing application-visible runtime state may depend on file system state that is restored by replay. Second, application-visible runtime state (e.g., file descriptors and offsets) is reconstructed from the tracking tables, allowing subsequent operations to observe a consistent runtime state. Finally, requests that were in progress at the time of the crash are re-executed and completed transparently. Because these requests remain pending in the shared-memory queues, the standby process can complete them through the normal request-processing path, without requiring special handling in LibFS. After these steps, applications can resume execution from their pre-crash state without requiring restart or manual repair.

After takeover, recovery operates on each domain's state independently. Because file system state is isolated across domains, these reconstruction and replay steps can proceed in parallel without mixing state across domain boundaries.

D. Resource Isolation

In current container stacks, file system resources such as metadata caches and I/O management are shared across all containers, leading to cross-container interference that is not fully controlled by existing mechanisms. One container can degrade another's metadata throughput or interfere with device bandwidth despite kernel-level isolation mechanisms. ConFS addresses this problem by enforcing resource isolation at both the in-memory state and device I/O levels, using container domains as the unit of isolation.

Domain-private Metadata and Buffering. ConFS prevents cross-domain interference by making all in-memory file system state domain-private, including both metadata structures and buffered data.

Metadata isolation. In conventional file systems, the dentry cache and inode cache are globally maintained structures. A container that generates intensive metadata operations can increase traversal and contention overhead in the shared cache, degrading metadata throughput for other containers regardless of any cgroup limit. This is because these caches are organized as global lookup structures over a shared directory hierarchy, where all containers contend on the same metadata namespace and synchronization paths, introducing cross-container contention along shared metadata access paths.

Inside FSServer, each domain maintains its own dentry cache, inode cache, and lookup structures. Instead of a single

global metadata structure, ConFS partitions metadata into per-domain instances, each managing its own directory hierarchy and lookup structures. As a result, metadata operations no longer contend on shared lookup paths, and intensive metadata activity in one domain cannot slow down metadata access in others.

Data buffering and writeback isolation. In conventional systems, buffering and writeback are globally managed within the kernel, coupling write behavior across containers. While mechanisms such as cgroups can limit resource usage at the container level, they do not decouple the underlying buffering and writeback paths, leaving cross-container interference largely unaddressed. This coupling arises because writeback decisions are made over shared buffering state and global control paths, allowing write activity from one container to influence when and how data from others is flushed. For example, the kernel uses a global dirty-page threshold to trigger background writeback: when one container drives the dirty ratio past this threshold, writeback is initiated system-wide, consuming device bandwidth and reducing the effective throughput of others.

ConFS eliminates this coupling by keeping buffered data private to each domain in userspace memory, independent of the kernel page cache. Writes accumulate in the buffer of the issuing domain and are flushed based on per-domain thresholds and policies. As a result, writeback is scoped to each domain, preventing one domain's write activity from triggering or affecting writeback behavior in others.

Domain-accounted I/O Control. After eliminating cross-domain contention in metadata access and writeback paths, ConFS must also regulate access to the shared storage device itself. This control is enabled by the fact that all I/O requests are issued through domain-local file system paths, allowing each request to be precisely attributed to its originating domain.

To implement this control, ConFS maintains a per-domain token bucket to regulate bandwidth consumption, which is refreshed periodically according to configured limits. If a domain exhausts its allocation, its worker delays further submissions until tokens become available.

Because all I/O requests, including both foreground and background operations, such as writeback, are consistently attributed to their originating domain, they can be regulated at submission time. As a result, no domain can exceed its configured bandwidth limit, and cross-domain interference at the device level is significantly reduced.

E. Implementation Issues

ConFS is implemented in about 5K LoC of C++ on Linux. The current prototype targets Linux and a single NVMe SSD managed by *Storage Performance Development Kit* (SPDK) [28]. Each domain maintains its own container image files to confine file accesses within the local domain state. To reduce resource redundancy, cross-domain sharing of image files can be enabled by relaxing isolation guarantees and permitting read-only system-call access to host-side image files. The implementation focuses on three mechanisms that directly support the core design of ConFS: shared-memory management for communication and recovery, worker management for shared execution with domain isolation, and device-space management for multiplexing storage across domains.

Shared Memory Allocation. ConFS bounds the amount of shared memory reserved for each domain to prevent unbounded growth while preserving domain-level memory isolation. In addition, it organizes shared memory into per-domain regions to ensure that metadata, request-tracking state, and recovery structures are not shared across domains. The shared memory is used for LibFS-FSServer communication and to support fast recovery.

In our prototype, the shared-memory threshold is configured at startup and defaults to 20 MiB, which can be increased based on the expected workload of the domain. This threshold bounds per-domain memory usage while keeping the communication and recovery path unchanged. When the threshold is reached, ConFS reclaims space through the existing sync and checkpoint path.

Worker Thread Management. FSServer uses a bounded pool of worker threads, with active workers pinned to dedicated cores. Workers are shared across domains: a single worker may serve operations from multiple domains, and multiple workers may serve operations within the same domain. This design allows CPU resources to be shared while relying on domain-local file system state, rather than dedicated execution resources per domain, to preserve isolation. Correctness relies on fine-grained locking over private metadata and runtime state within each domain, allowing multiple threads to execute file system operations concurrently.

In the current prototype, queues are statically grouped and each worker polls an assigned group in round-robin order. This is the scheduling policy used in the prototype and does not affect the isolation guarantees. When many domains coexist, the number of queues can grow and increase polling overhead. More adaptive worker scheduling, as explored in recent userspace storage systems [29], [30], is orthogonal to our isolation design and left as future work.

Device Space Management. To allow multiple domains to share a single SSD without static partitioning, ConFS divides the device into fixed-size segments (2GiB in our prototype) and manages them using a buddy-style allocator. Each domain is assigned a disjoint set of segments, ensuring that data placement and I/O remain isolated across domains.

Within each domain, file system operations use domain-relative logical block addresses. FSServer maintains a per-domain mapping table that translates these addresses to physical device locations and ensures that all I/O stays within the domain's allocated space.

When a new domain is created, FSServer allocates an initial set of segments from the shared pool (2 GiB by default, configurable based on domain requirements) and initializes a per-domain file system. This organization allows domains to share the device without static partitioning while preserving physical separation in space allocation. The current prototype

TABLE I: Representative CVEs discussed in the access-isolation analysis.

Failure type	CVE
Post-validation Access Redirection	CVE-2018-15664 [24] CVE-2019-10152 [33] CVE-2019-18466 [34] CVE-2023-0778 [35]
Mount-Target Redirection	CVE-2017-1002101 [36] CVE-2021-25741 [37] CVE-2021-30465 [38] CVE-2022-23648 [39] CVE-2025-31133 [40]

allocates segments only at domain creation; supporting later growth by attaching additional segments is left as future work.

IV. Evaluation

In this section, we evaluate ConFS by answering the following questions.

- How does ConFS provide access isolation between different containers?
- How does ConFS provide fault isolation for containerized applications? How quickly can ConFS recover from failures?
- How effectively does ConFS provide resource isolation and stable performance?
- Does ConFS achieve performance comparable to that of both kernel file systems and userspace file systems?

Experimental setup. We run all experiments on a single server equipped with a 96-core Intel Xeon Platinum 8488C CPU and 512 GB of DRAM. We use a 375 GB Intel Optane P4800X SSD for its ultra-low device access latency ($\sim$7 μs), which makes software stack overhead more prominent relative to device access time. The server runs Ubuntu 20.04 with Linux kernel 5.4, and the userspace storage driver uses SPDK 18.04. We disable hyperthreading, turbo boost, and power-saving features to reduce measurement noise, following established benchmarking practices in previous works [30], [31].

Compared Systems. We compare ConFS against Ext4 [20] and F2FS [32] as representative kernel local file systems. In all experiments, Ext4 and F2FS are deployed as a single shared file system instance across containers, reflecting the standard deployment of containers in which co-located containers share the host kernel file system. We also compare against uFS [29], a high-performance userspace baseline without container-aware isolation. For recovery experiments, we additionally implement an Ananke-style [15] recovery mechanism on top of uFS. Microbenchmarks use direct I/O to expose storage-path overhead rather than page-cache effects. Application and recovery experiments use each file system's normal cache behavior. Unless otherwise noted, each benchmarked application runs in a single container domain, and each experiment is repeated 5 times and we report the average result.

A. Analysis of Access Isolation

To evaluate the effectiveness of ConFS's access-isolation design, we analyze publicly disclosed container-escape vulnerabilities from 2017 to 2025 whose attack paths rely on shared host file system metadata (Table I). These vulnerabilities span diverse file system interfaces, such as file copy, volume mount, and image configuration.

Despite differences in surface operations, these vulnerabilities share a common root cause: multiple containers operate on the same file system metadata and directory hierarchy on the host. As a result, path resolution and metadata traversal are performed over shared state, allowing one container to influence or redirect another container's access. In other words, access semantics are determined by a shared execution path rather than being confined within container boundaries.

For example, CVE-2018-15664 illustrates the risk of a shared execution path. During a `docker cp` operation, the source pathname may first be validated within the container file system, for example as `/tmp/data/etc/shadow`. However, before the copy is actually performed, an attacker can replace `/tmp/data` with a symbolic link to `/`. As a result, when the docker daemon later accesses the same pathname, it is re-resolved in the host file system and points to `/etc/shadow` on the host. This attack is possible because the check and the use both depend on shared file system metadata, allowing the execution path to be altered between the two operations.

ConFS blocks this class of attacks by removing shared host metadata from cross-domain execution paths. It uses the container domain as the unit of access isolation and binds each request to its domain through trusted communication channels. FSServer then resolves and executes each request using only the domain's own metadata, both in memory and on disk, so that pathname resolution and metadata access are determined by domain-local state rather than shared global metadata. Containers that intentionally share data are explicitly grouped into the same domain, making sharing a controlled property of the isolation boundary.

Although FSServer retains a small amount of shared runtime state, such as worker threads and allocation structures, this state does not participate in pathname resolution or metadata traversal and therefore does not affect the semantic target of a request. As a result, for file system operations served by FSServer, the shared-host-metadata dependency exploited by these escape attacks is removed. We reproduced representative exploits shown in Table I and confirmed that ConFS blocks the corresponding escape paths in all tested cases. For each CVE, we verified the exploit under the original vulnerable kernel configuration, then confirmed that the same exploit fails under ConFS because the shared host metadata path required by the attack is no longer available.

B. Fault Isolation and Recovery

We evaluate fault isolation and recovery in two steps. First, we measure end-to-end application performance during a file system crash to show that ConFS enables fast and transparent

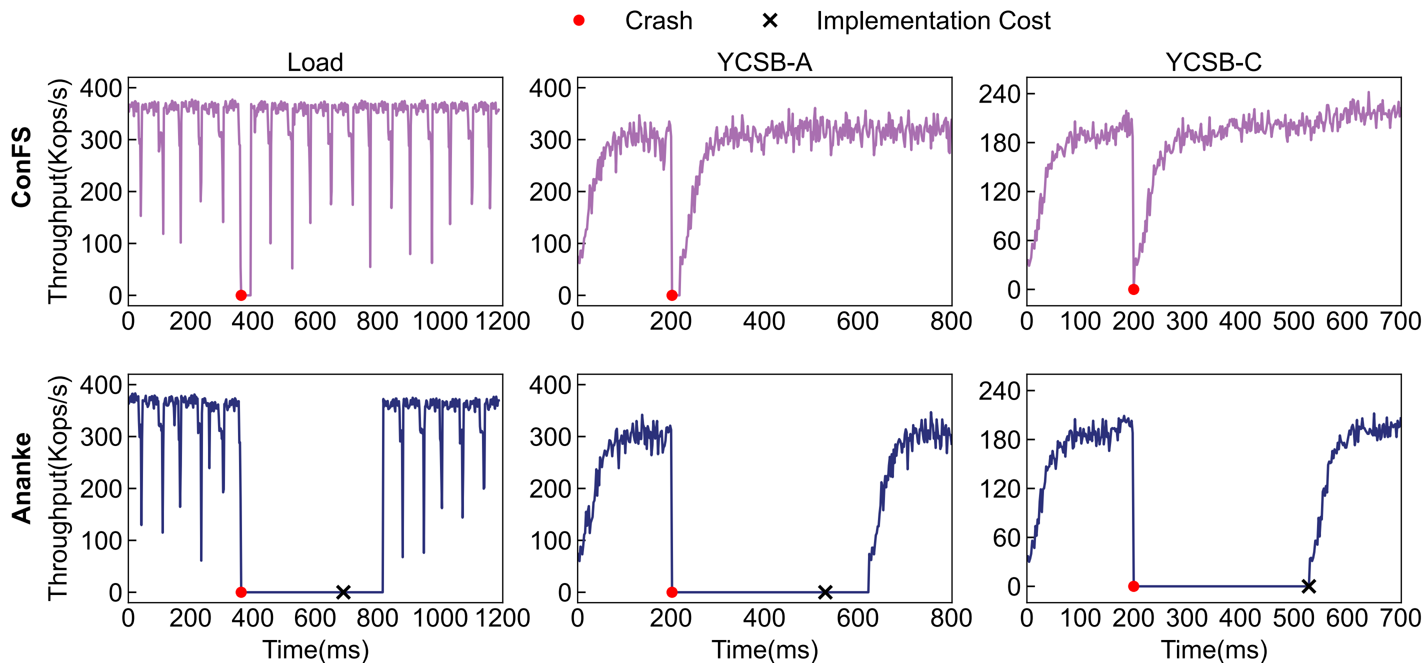

Fig. 5: Fault isolation with ConFS and Ananke.

recovery with minimal disruption. Second, we break down the recovery process into R1, R2, and R3 to understand how each component contributes to the overall recovery latency.

Effective isolation of file system fault We evaluate whether ConFS provides fast and transparent recovery with minimal disruption to running applications. We run LevelDB [41] on ConFS with `YCSB-Load`, `YCSB-A`, and `YCSB-C` to cover write-only, mixed, and read-dominant workloads, respectively. We inject a file system crash during execution and measure throughput over time to evaluate the impact of the crash and subsequent recovery. Kernel file systems (Ext4 and F2FS) require a full remount or system reboot after a crash, while uFS does not reconstruct the runtime state needed by applications that were in progress at crash time. As a result, none of these approaches support transparent application resumption after a crash. We therefore compare ConFS against Ananke, the only baseline that provides application-transparent recovery. To ensure a fair comparison, our Ananke-style implementation follows the same transparent-recovery goal on top of uFS, namely restoring file system service together with the runtime state needed by blocked applications to resume after a crash. Ananke is built on uFS and enables fast file system restart with state reconstruction, making it directly comparable to ConFS. We also compare the two systems phase by phase, so that fixed startup overhead can be separated from the recovery logic itself.

Figure 5 shows the throughput of LevelDB running on ConFS and Ananke. A file system crash occurs at the point marked by the red circle. ConFS resumes service within 30.2, 15.8, and 0.9 ms under `YCSB-Load`, `YCSB-A`, and `YCSB-C`, respectively, whereas Ananke requires 454.6, 420.2, and 328.7 ms. A large portion of Ananke's recovery time comes from the fixed startup overhead of its underlying uFS implementation (about 327.6 ms), corresponding to the interval between the red circle and the "x" marker in Figure 5. We report this cost explicitly for fairness, since it reflects the restart overhead inherited from the uFS base rather than the recovery logic itself. Even after excluding this cost, Ananke still requires 127.0, 92.6, and 1.1 ms, which remains slower than ConFS. This difference is due to ConFS's more selective recovery design. By using an in-memory shadow journal and incrementally maintained request state, ConFS avoids scanning and classifying all logged requests during recovery. The gap is most pronounced under `YCSB-Load`, where a larger amount of non-persisted state must be recovered.

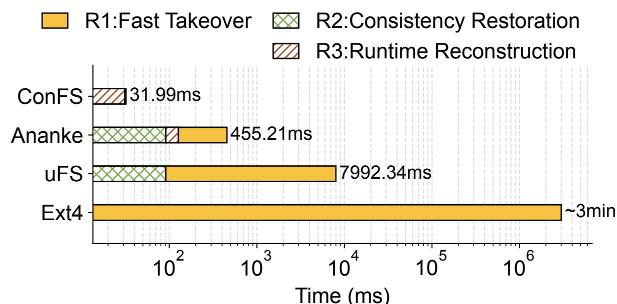

Fig. 6: Crash recovery breakdown.

Breakdown of file system recovery. We break down the recovery time of four file systems: ConFS, Ananke, uFS, and Ext4. We run LevelDB with the write-only `YCSB-Load` workload, which imposes a high recovery burden, and inject a crash

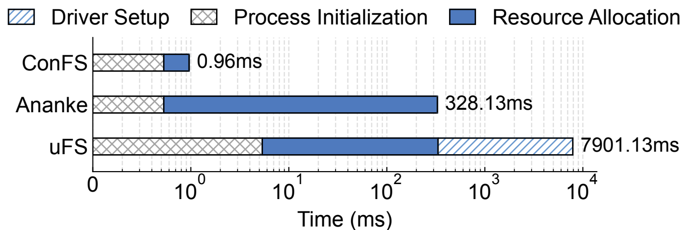

Fig. 7: Breakdown of process initialization.

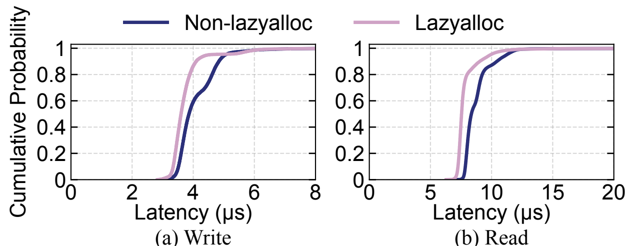

Fig. 8: Lazy allocation cost.

after 5 million operations. We decompose recovery into three phases: Fast Takeover (R1), Consistency Restoration (R2), and Runtime Reconstruction (R3). Overall, ConFS achieves significantly faster recovery by reducing overhead in all three phases: eliminating startup cost in R1, removing device I/O in R2, and avoiding full log replay in R3.

Figure 6 shows the results. ConFS, Ananke, uFS, and Ext4 recover in 31.92 ms, 455.17 ms, 7987.40 ms, and approximately 3 minutes, respectively. uFS and Ext4 only perform process startup and consistency restoration, as they do not reconstruct runtime state. In contrast, ConFS and Ananke implement the full R1–R3 recovery path.

Fast Takeover (R1): In R1, ConFS, Ananke, uFS, and Ext4 take 0.89 ms, 328.09 ms, 7896.19 ms, and approximately 3 minutes, respectively. Ext4 incurs the highest cost due to requiring a full OS reboot. uFS, Ananke, and ConFS are all user-level file systems built on SPDK. Figure 7 further breaks down their R1 time into process recreation, driver initialization, and resource allocation.

uFS requires 5.36 ms for process creation and 7568.17 ms for driver initialization, as it must recreate the file system process and reinitialize SPDK from scratch. In contrast, ConFS and Ananke complete these steps in only 0.6 ms by using a pre-spawned standby process that has already finished initialization before the crash.

For resource allocation, ConFS requires only 0.43 ms, whereas uFS and Ananke require 327.6 ms due to large upfront memory allocation. Specifically, uFS and Ananke preallocate a large number of fine-grained memory objects at startup, which increases initialization latency. ConFS instead adopts a lazy allocation strategy, creating objects only on demand. This significantly reduces recovery-time overhead. To evaluate the impact of lazy allocation, we measure the latency of accessing newly allocated objects. Figure 8 shows the cumulative distribution of read and write latency with and

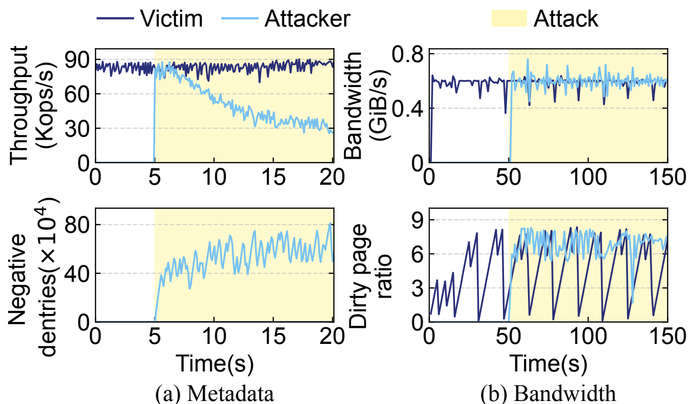

Fig. 9: Resource interference test on ConFS.

without lazy allocation. Lazy allocation adds only 1–2 μs to the first access of a new object and incurs no additional overhead thereafter.

Consistency Restoration (R2): In R2, Ext4 takes 13.40 ms to recover and read journal entries during remount. Ananke and uFS take 91.21 ms, including 7.5 ms for reading valid journal entries and 83.2 ms for scanning over 50,000 blocks in the journal region. In contrast, ConFS reduces this time to 0.27 ms using a shadow journal in shared memory. After takeover, the standby process reads only the on-device journal header to identify the durable prefix and replays it directly from the in-memory shadow journal. This avoids journal-block reads from the device while preserving ordered journaling semantics.

Runtime Reconstruction (R3): In R3, Ananke requires 35.87 ms because it must iterate through all logged requests and classify each one during recovery. ConFS completes R3 in 30.76 ms by reconstructing runtime file state directly from the tracking table and replaying only the necessary operations. Specifically, ConFS selectively replays requests whose per-inode dirty bits indicate unsynchronized in-memory changes, while avoiding re-execution of operations whose updates have already reached stable storage. This eliminates both the overhead of scanning the full log and the redundancy of replaying unnecessary operations.

C. Resource Isolation

We evaluate whether ConFS eliminates cross-container interference and provides stable performance under contention.

First, we reproduce the metadata interference experiment described in Section II-D. A victim container runs a metadata-intensive workload, while an attacker continuously creates and deletes files to generate a large number of negative dentries. As shown in Figure 9, the victim sustains a stable throughput of 87 Kops/s throughout the experiment, even after the attacker starts generating negative dentries. This is because ConFS isolates metadata state across containers: dentries created by the attacker remain within its own file system context and do not affect the victim.

Next, we evaluate interference along the I/O path. Although *cgroups* can regulate both foreground and background I/O at

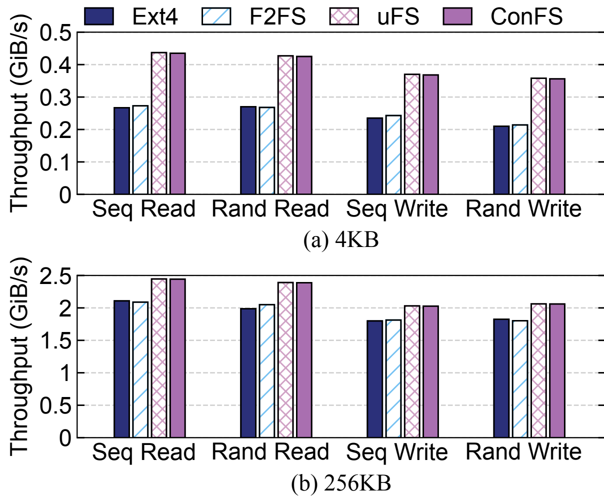

Fig. 10: Microbenchmark throughput.

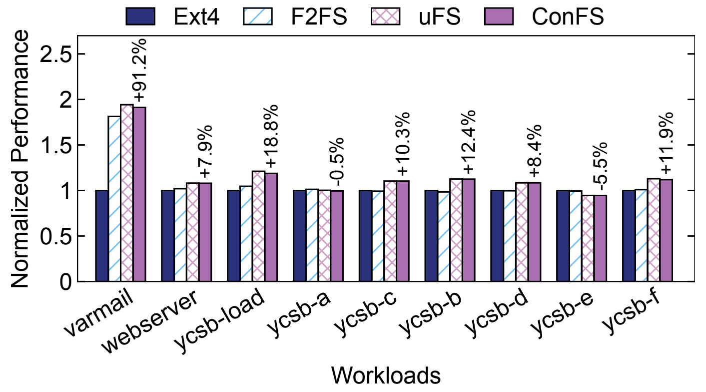

Fig. 11: Application throughput.

the device level, containers in conventional systems still share the underlying dirty-page accounting and writeback-triggering mechanisms. As a result, one container can indirectly affect another by increasing global dirty-page pressure and causing more frequent writeback activity. We run a victim container executing a mixed read-write workload and an attacker container continuously generating dirty pages. Both containers are limited to 0.6 GiB/s using *cgroups*. As shown in Figure 9, ConFS maintains stable bandwidth at 0.6 GiB/s throughout the experiment. This is because ConFS isolates buffered data and writeback management across containers, preventing one container from influencing the I/O behavior of others.

These results show that ConFS effectively prevents metadata interference and writeback interference under the tested scenarios, by isolating file system state and management across container domains.

D. Comparable Performance

We evaluate whether ConFS provides enhanced isolation and fast recovery with negligible performance overhead. We measure both microbenchmark performance to quantify the base overhead of the I/O path, and application-level performance to assess its impact under realistic workloads.

Microbenchmarks. To quantify the base I/O overhead of ConFS, we run sequential and random read/write microbenchmarks at 4 KB and 256 KB request sizes. Overall, ConFS achieves higher throughput than kernel file systems while maintaining performance comparable to uFS, showing that its additional isolation and recovery support adds negligible overhead on the common I/O path.

At the 4 KB granularity (Figure 10a), ConFS improves both read and write throughput by $1.6\times$ to $1.7\times$ over Ext4 and F2FS. This improvement comes from its userspace I/O stack, which eliminates kernel overhead such as system calls, VFS processing, and interrupt handling. For example, in 4 KB random read, ConFS achieves an average latency of $9.19\,\mu$s, including about $7\,\mu$s of device latency (close to the hardware limit of the Optane P4800X) and only $\sim 2\,\mu$s of software overhead. In contrast, the random read latencies of Ext4 and F2FS are $14.47\,\mu$s and $14.58\,\mu$s, respectively, with an additional $\sim 7\,\mu$s overhead introduced by the kernel software stack (including system calls, VFS layer processing, and interrupt handling).

At the 256 KB request size (Figure 10b), ConFS achieves up to $1.20\times$ and $1.14\times$ higher throughput than Ext4 and F2FS. The performance gap narrows compared to the 4 KB case, as larger I/O sizes amortize software overhead and make device access the dominant factor.

Compared to uFS, ConFS shows less than a 1% performance difference across all patterns and request sizes. Since both systems share a similar SPDK-based userspace I/O stack, this result indicates that the isolation and recovery mechanisms added by ConFS introduce almost no additional cost beyond the baseline userspace I/O path.

Application Workloads. We evaluate the application-level performance of ConFS using Filebench (`varmail` and `webserver`) and YCSB workloads on LevelDB. Overall, ConFS achieves comparable or better performance than kernel file systems, demonstrating that enhanced isolation introduces negligible overhead in realistic workloads. Figure 11 reports throughput normalized to Ext4.

For write-intensive workloads, ConFS outperforms kernel file systems. In the `varmail` test, ConFS achieves $1.91\times$ the throughput of Ext4 and $1.06\times$ that of F2FS. This improvement is largely due to frequent `fsync` operations in `varmail`. Ext4 relies on a global journaling mechanism (JBD2), which results in an average `fsync` latency of $101\,\mu$s. F2FS adopts a log-structured design that converts random updates into sequential writes, improving `fsync` performance, but it still incurs kernel software overhead. In contrast, ConFS combines a userspace I/O stack with a per-inode logging design, reducing the average `fsync` latency to $33\,\mu$s.

A similar trend is observed in YCSB workloads. For `YCSB-Load` (pure insert), `YCSB-B` (5% update), `YCSB-D` (5% update), and `YCSB-F` (read-modify-write), ConFS improves throughput by $1.1\times$ to $1.2\times$ over Ext4. This is because ConFS bypasses the kernel storage stack and interacts directly

with the NVMe device via SPDK, reducing software overhead along the data flushing path.

For read-intensive workloads, ConFS achieves performance comparable to Ext4. It delivers 1.10× and 1.08× the throughput of Ext4 for `YCSB-C` (pure read) and `webserver`, respectively. For `YCSB-A` (50% read) and `YCSB-E` (95% scan), ConFS achieves 0.99× and 0.95× of Ext4 performance. This is because Ext4 and F2FS benefit from highly optimized page cache and readahead mechanisms that effectively hide backend I/O latency. These mature optimizations give kernel file systems an advantage in sequential scan workloads such as `YCSB-E`, while ConFS remains competitive overall.

Compared to uFS, ConFS shows less than a 2% performance difference across all workloads. In write-intensive cases such as `YCSB-Load` and `varmail`, the throughput reduction is only 1.9% and 1.5%, respectively, and for all other workloads the gap remains below 1%. Since both systems share a similar SPDK-based userspace I/O stack, this result confirms that the additional mechanisms for isolation and recovery incur only negligible overhead under realistic application workloads.

Overall, these results show that ConFS stays close to the uFS baseline while remaining competitive with, and often outperforming, kernel file systems. Taken together, the results indicate that enhanced isolation and fast recovery are achieved with negligible performance overhead.

V. Related Work

Container file systems and storage systems. Prior work [19], [22], [42] on container storage mainly optimizes image layering and I/O efficiency on top of shared host file systems. Container file systems are typically built by stacking read-only image layers with a writable layer through union file systems such as OverlayFS [9] and AUFS [10]. BAOverlay [19] improves layered storage for write-intensive workloads by making overlay files block-accessible. CAST [22] reduces the overhead of the shared host I/O stack for multi-container accesses. Diciclo [42] improves performance stability by serving different tenants with separate user-level storage clients. These systems improve container storage performance and efficiency, but they still treat the host file system as the sharing boundary and do not align file system execution or state with container-level isolation boundaries.

Isolation and recovery in container file systems. For access isolation, Patrol [12] and PACED [13] prevent container-escape and boundary violations by adding checks to the host file system path, but the file system state used for pathname resolution and metadata lookup is still shared across containers. Stronger-isolation runtimes such as Kata Containers [43] and gVisor [44] reduce the kernel attack surface by interposing a VM or a userspace kernel between applications and the host. However, at the file system level, shared volumes still traverse the host kernel file system—for example, via virtio-fs in Kata Containers, which relays guest file system requests through the host VFS—leaving file system vulnerabilities in shared-volume scenarios unmitigated [12], [45]–[47]. For fault isolation, IceFS [48] separates file system state into smaller independent units so that a failure in one unit does not affect others. Membrane [18] and Shadow Filesystems [16] make kernel file systems restartable or recoverable through standby replicas, and Ananke [15] provides transparent recovery for userspace file system services. These systems reduce the scope or cost of failures, but their failure boundaries are not aligned with the intended sharing and isolation boundary of containers [15], [16], [18]. For resource isolation, prior work has focused on isolating CPU [49], [50], memory [51], storage traffic [26], [48], and network bandwidth [52], [53], and on strengthening cgroups-based resource control [54], [55]. At the file system level, Trātr [17] shows that shared kernel data structures can be exploited to amplify cross-tenant interference, and Diciclo [42] and CAST [22] reduce contention by separating parts of the storage path at the user level, but the underlying kernel file system state that causes interference is still shared. ConFS partitions file system state at the container-domain level, so that access isolation, fault isolation, and resource isolation share the same boundary.

Userspace file systems and storage stacks. Userspace file systems provide the architectural basis for ConFS. In-process file systems such as Strata [56], SplitFS [57], and Aeolia [30] run file system logic inside the application process, so enforcing isolation across containers would require additional protection mechanisms between processes that share the same file system state. Device-level file systems such as DevFS [58] and CrossFS [59] place file system logic in device firmware, so container-level isolation would depend on hardware partitioning mechanisms that current devices do not provide at the required granularity. Userspace daemon file systems such as uFS [29] run the file system as a separate trusted process, which naturally provides a control point for enforcing isolation, but uFS does not partition internal file system state or failure domains according to container-level sharing boundaries. ConFS builds on this daemon architecture and extends it to make the container domain the file system isolation boundary.

VI. Conclusion

This paper presents ConFS, a container-aware userspace file system that reorganizes file system isolation around container domains. By aligning file system execution and state management with domain boundaries, and by reconstructing file system state on a per-domain basis after an FSServer failure, ConFS provides enhanced access, fault, and resource isolation in containerized environments. Evaluation on NVMe SSDs shows that ConFS achieves enhanced isolation with less than 2% performance degradation compared to existing userspace file systems and up to 1.6× higher performance than Ext4.

VII. Acknowledgements

We sincerely thank the anonymous reviewers for their insightful feedback. This work is supported by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92270202).

REFERENCES

[1] "Docker," https://www.docker.com.

[2] "Podman," https://podman.io.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, 2016.

[4] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: Workload Autoscaling at Google," in *Proc. of EuroSys*, 2020.

[5] Q. Liu and Z. Yu, "The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: A View from Alibaba Trace," in *Proc. of ACM SOCC*, 2018.

[6] "namespaces(7) - Linux manual page." https://man7.org/linux/man-pages/man7/namespaces.7.html.

[7] T. Heo, "Control Group v2," https://www.kernel.org/doc/html/v6.1/admin-guide/cgroup-v2.html.

[8] "mount_namespaces(7) - Linux manual page." https://man7.org/linux/man-pages/man7/mount_namespaces.7.html, 2025.

[9] Docker Inc., "Use the OverlayFS storage driver," https://docs.docker.com/storage/storagedriver/overlayfs-driver, 2020.

[10] J. R. Okajima, "Linux AuFS Examples: Another Union File System Tutorial," http://aufs.sourceforge.net/aufs2.

[11] Docker Inc., "Bind mounts," https://docs.docker.com/storage/bind-mounts.

[12] Z. Li, W. Liu, X. Wang, B. Yuan, H. Tian, H. Jin, and S. Yan, "Lost along the Way: Understanding and Mitigating Path-Misresolution Threats to Container Isolation," in *Proc. of ACM CCS*, 2023.

[13] M. Abbas, S. Khan, A. Monum, F. Zaffar, R. Tahir, D. Eyers, H. Irshad, A. Gehani, V. Yegneswaran, and T. Pasquier, "PACED: Provenance-based Automated Container Escape Detection," in *Proc. of IEEE IC2E*, 2022.

[14] Z. Jian and L. Chen, "A Defense Method against Docker Escape Attack," in *Proc. of ICCSP*, 2017.

[15] J. Liu, Y. Dai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Fast, Transparent Filesystem Microkernel Recovery with Ananke," in *Proc. of USENIX FAST*, 2025.

[16] J. Liu, X. Hao, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and T. Chajed, "Shadow Filesystems: Recovering from Filesystem Runtime Errors via Robust Alternative Execution," in *Proc. of ACM HotStorage*, 2024.

[17] Y. Patel, C. Ye, A. Sinha, A. Matthews, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Using Trāṭr to tame Adversarial Synchronization," in *Proc. of USENIX Security*, 2022.

[18] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Membrane: Operating System Support for Restartable File Systems," *ACM Transactions on Storage (TOS)*, 2010.

[19] Y. Sun, J. Lei, S. Shin, and H. Lu, "Baoverlay: a Block-accessible Overlay File System for Fast and Efficient Container Storage," in *Proc. of ACM SoCC*, 2020.

[20] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The New Ext4 Filesystem: Current Status and Future Plans," in *Proc. of the Linux Symposium*, 2007.

[21] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. of USENIX ATC*, 1996.

[22] S. Wu, Z. Huang, P. Chen, H. Fan, S. Ibrahim, and H. Jin, "Container-aware I/O Stack: Bridging the Gap between Container Storage Drivers and Solid State Devices," in *Proc. of ACM VEE*, 2022.

[23] Y. Liu, H. Li, Y. Lu, Z. Chen, and M. Zhao, "An Efficient and Flexible Metadata Management Layer for Local File Systems," in *Proc. of IEEE ICCD*, 2019.

[24] "CVE-2018-15664," https://nvd.nist.gov/vuln/detail/cve-2018-15664.

[25] "CVE-2019-19319," https://nvd.nist.gov/vuln/detail/cve-2019-19319.

[26] T. Heo, D. Schatzberg, A. Newell, S. Liu, S. Dhakshinamurthy, I. Narayanan, J. Bacik, C. Mason, C. Tang, and D. Skarlatos, "IOCost: Block IO Control for Containers in Datacenters," in *Proc. of ACM ASPLOS*, 2022.

[27] Kubernetes, "Pods," https://kubernetes.io/docs/concepts/workloads/pods.

[28] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A Development Kit to Build High Performance Storage Applications," in *Proc. of IEEE CloudCom*, 2017.

[29] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Scale and Performance in a Filesystem Semi-Microkernel," in *Proc. of ACM SOSP*, 2021.

[30] C. Li, R. Yi, Z. Zhang, J. Liu, C. Min, J. Zhang, Y. Luo, X. Wang, Z. Wang, and D. Zhou, "Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack," in *Proc. of ACM SOSP*, 2025.

[31] S. Yadalam, C. Alverti, V. Karakostas, J. Gandhi, and M. Swift, "BypassD: Enabling Fast Userspace Access to Shared SSDs," in *Proc. of ACM ASPLOS*, 2024.

[32] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proc. of USENIX FAST*, 2015.

[33] "CVE-2019-10152," https://nvd.nist.gov/vuln/detail/cve-2019-10152.

[34] "CVE-2019-18466," https://nvd.nist.gov/vuln/detail/cve-2019-18466.

[35] "CVE-2023-0778," https://nvd.nist.gov/vuln/detail/cve-2023-0778.

[36] "CVE-2017-1002101," https://nvd.nist.gov/vuln/detail/cve-2017-1002101.

[37] "CVE-2021-25741," https://nvd.nist.gov/vuln/detail/cve-2021-25741.

[38] "CVE-2021-30465," https://nvd.nist.gov/vuln/detail/cve-2021-30465.

[39] "CVE-2022-23648," https://nvd.nist.gov/vuln/detail/cve-2022-23648.

[40] "CVE-2025-31133," https://nvd.nist.gov/vuln/detail/cve-2025-31133.

[41] Google, "LevelDB," https://github.com/google/leveldb.

[42] G. Kappes and S. V. Anastasiadis, "Diciclo: Flexible User-level Services for Efficient Multitenant Isolation," *ACM Transactions on Computer Systems (TOCS)*, vol. 42, 2024.

[43] "Kata Containers," https://katacontainers.io/.

[44] "gVisor: The Container Security Platform," https://gvisor.dev/.

[45] K. Manakkal, N. Daughety, M. Pendleton, and H. Lu, "LITESHIELD: Secure Containers via Lightweight, Composable Userspace μKernel Services," in *Proc. of USENIX ATC*, 2025.

[46] T. Miemietz, V. Reusch, M. Hille, L. Wrenger, J. Eisoldt, J. Klötzke, M. Kurze, A. Lackorzynski, M. Roitzsch, and H. Härtig, "MettEagle: Costs and Benefits of Implementing Containers on Microkernels," in *Proc. of USENIX OSDI*, 2025.

[47] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds," in *Proc. of IEEE DSN*, 2017.

[48] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical Disentanglement in a Container-Based File System," in *Proc. of USENIX OSDI*, 2014.

[49] C. A. Waldspurger and W. E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," in *Proc. of USENIX OSDI*, 1994.

[50] S. Das, V. R. Narasayya, F. Li, and M. Syamala, "CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-Service," *Proc. of the VLDB Endowment*, vol. 7, no. 1, 2013.

[51] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002.

[52] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for {Multi-Tenant} Cloud Storage," in *Proc. of USENIX OSDI*, 2012.

[53] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, and C. Kim, "{EyeQ}: Practical Network Performance Isolation for the Multi-tenant Cloud," in *Proc. of USENIX HotCloud*, 2012.

[54] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's Escape: Breaking the Resource Rein of Linux Control Groups," in *Proc. of ACM CCS*, 2019.

[55] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma *et al.*, "Demons in the Shared Kernel: Abstract Resource Attacks against OS-level Virtualization," in *Proc. of ACM CCS*, 2021.

[56] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proc. of ACM SOSP*, 2017.

[57] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory," in *Proc. of ACM SOSP*, 2019.

[58] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a True Direct-Access File System with DevFS," in *Proc. of USENIX FAST*, 2018.

[59] Y. Ren, C. Min, and S. Kannan, "CrossFS: A Cross-layered Direct-Access File System," in *Proc. of USENIX OSDI*, 2020.