A Zero-Trust Block Store with TEE-Issued, Kernel-Enforced Immutability

Johan B

December 10, 2025

Abstract

Modern ransomware and data-theft campaigns frequently culminate in attackers obtaining powerful credentials or a root shell on storage servers and then encrypting, deleting, or silently modifying large volumes of data. This paper presents a design for a zero-trust block store that assumes such host compromises and still enforces immutability of committed ciphertext and a strict separation between mutation and decryption authority. A small authorizer running in a trusted execution environment (TEE) issues short-lived, one-use capabilities for destructive operations such as overwrites and deletions, while standard file creation and growth remain on a low-latency fast path governed by kernel access controls. A Linux Security Module (LSM) in the kernel verifies these capabilities, derives resource identities inside the kernel, and attaches latches bound to specific files and byte ranges, with a limited budget, so that only bounded changes to those ranges can succeed. Data is encrypted at the client with AEAD and attribute-based encryption (ABE); storage nodes hold ciphertext and ABE-protected headers only and never receive tenant root keys or data-encryption keys. Replication uses chain replication per placement group, with the tail defining commit and triggering sealing of the affected byte ranges on all replicas. The design targets XFS without reflink, optional DRBD in synchronous Protocol C mode for per-replica volume mirroring to a hot standby, and SELinux hardening, and is presented as a concrete architecture and threat model for future implementation and evaluation.

1 Introduction

Ransomware and large-scale data theft have become routine, with attackers frequently moving laterally through networks until they reach the critical aggregation points: storage services, backup controllers, and hypervisor datastores. Whether initial access is gained via exposed gateways, compromised identity providers, or stolen credentials, the endpoint of these campaigns is often the same: for a critical window of time, the attacker possesses the credentials necessary to act as a legitimate operator.[5, 10, 12]

The fundamental vulnerability in modern storage is that possession of a root shell or privileged service account automatically confers total authority over the data. This failure stems from an architectural reliance on implicit trust: traditional storage stacks assume that any request reaching the storage node—having passed network-level access controls—is valid. In this perimeter-dependent model, the system lacks the internal boundaries to distinguish between a legitimate administrator and an intruder. Consequently, a single host compromise grants an attacker the unchecked ability to overwrite, truncate, or clone any file on attached volumes, and to exfiltrate plaintext if keys are present.

This paper proposes a design for a zero-trust block store that addresses this flaw by separating mutation authority from host privilege. We operate under the assumption that host compromise is inevitable and argue that storage nodes must be treated as untrusted user space wrapping a hardened trusted computing base (TCB). By enforcing strict separation of duties, we ensure that obtaining administrative privileges on a host does not grant the ability to rewrite or decrypt committed data.

The core of this solution is the migration of mutation authority out of the host OS and into a Trusted Execution Environment (TEE). A small, isolated authorizer issues short-lived, one-use capabilities for sensitive operations—such as edits to sealed content or administrative overrides. These capabilities are enforced not by the untrusted user-space daemon, but by a Linux Security Module (LSM) deep within the kernel. The LSM acts as a cryptographic gatekeeper: it verifies TEE-signed capabilities, derives resource identities from immutable filesystem attributes, and installs a "latch" that permits only a specific, bounded window of write activity. Operations that fall outside this latched budget, or that lack a valid capability, are denied at the system call level, regardless of the user's privilege level.[1, 2]

To secure data at rest, we define immutability through a "seal-after-commit" protocol aligned with chain replication. Data is encrypted at the client using a Tenant Root Key (TRK) that never resides on storage nodes. Writes are sent to a replication head and forwarded to a tail; the tail validates a BLAKE3 content commitment and defines the commit point. Upon tail acknowledgement, all replicas mark the affected byte ranges as sealed. Once sealed, these ranges become immutable to all standard write paths—including direct I/O and memory mapping—unless a specific TEE capability is presented to unlock them for a single transaction.

This work details the architecture and threat model for such a system, targeting a concrete implementation on XFS without reflink, utilizing optional DRBD mirroring for local durability, and leveraging SELinux to confine the host environment.

2 Threat Model

This section explains what the system is designed to withstand and the core security requirements derived from the assumed-breach premise.

Attacker capabilities. The attacker is allowed to gain control of user space on one or more storage nodes and to escalate to administrative privileges (root). In this state, they can run arbitrary binaries, read local disks, open network connections, invoke system calls, and steal resident service credentials.[5] However, they cannot modify the Secure Boot and lockdown-verified kernel, load unsigned modules, or read kernel memory.[18] They cannot forge authorizer signatures unless the TEE is compromised. The design assumes host compromise is inevitable and aims to bound the window of authorized mutation available to an attacker.

Network model. The network is untrusted; messages may be reordered, duplicated, or delayed. Freshness and partial ordering must be enforced by the endpoints. We assume a bounded window of write-like activity is acceptable if a valid capability is presented, but indefinite replay must be prevented.

Trusted computing base (TCB). The TCB is restricted to the boot-verified Linux kernel, the SELinux policy, the authorization LSM, kernel keyrings, and the authorizer's TEE.[18, 23] The orchestrator, DRBD, and the storage daemon are explicitly outside the TCB.

Replay and Topology Requirements. To resist replay attacks across network topology changes or node restarts, the system relies on two monotonic properties. First, a *placement-group epoch* must fence off capabilities issued for prior topology configurations. Second, a hardware-backed monotonic counter (e.g., TPM NVRAM) must fence off capabilities issued for prior boots of the same node, ensuring that a rebooted node effectively enters a new security context.

3 Design Overview

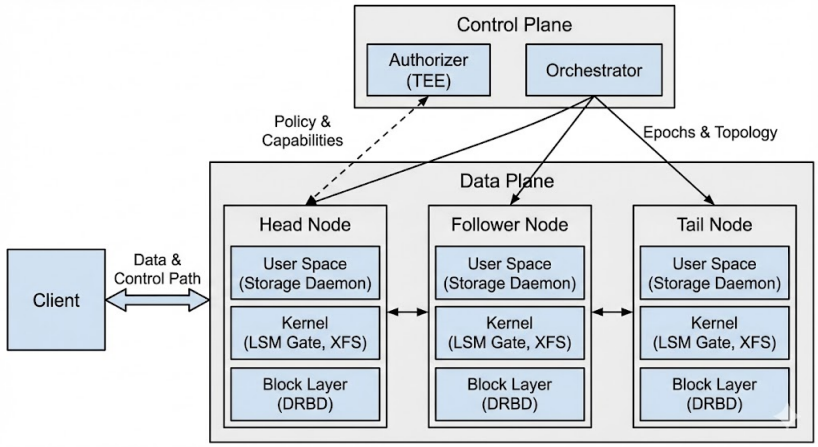

Figure 1: System Architecture. The Control Plane (TEE Authorizer) is logically separated from the Data Plane. Clients authenticate with the Control Plane to receive capabilities, which are then enforced by the Kernel LSM on Storage Nodes.

3.1 Data Path, Replication, and Sealing

Each object is mapped to a placement group using a hash of its file or path identifier. For each placement group, the orchestrator records an ordered list of replicas, defining a logical chain with a head and a tail.[22] Clients send encrypted data to the head. The head writes locally and forwards the same encrypted data, along with the capability metadata, to the next replica. Intermediate replicas follow the same pattern: they verify the capability locally, write the encrypted data to their local stable storage, and then forward the request to the next node in the chain.

The tail has a special role. It verifies the content commitment for the frame, writes the data to stable storage, and only then sends an acknowledgement upstream. This acknowledgement is the commit point for that placement group. Once the head receives the acknowledgement from the tail, the write is considered committed. At that point, each replica marks the corresponding byte ranges as sealed. Sealing installs state that denies further modification of those ranges through all covered system calls and ioctls. Sealing is idempotent and can be replayed during recovery from a compact log of committed ranges or from filesystem metadata that records which segments have been committed; replicas that crash after commit but before sealing will re-seal the affected ranges when they rejoin. If the tail never acknowledges, for example because

it fails mid-write, no sealing occurs and the affected ranges can be overwritten or discarded during recovery. This commit-then-seal behavior aligns the immutability boundary with a clear replication boundary.

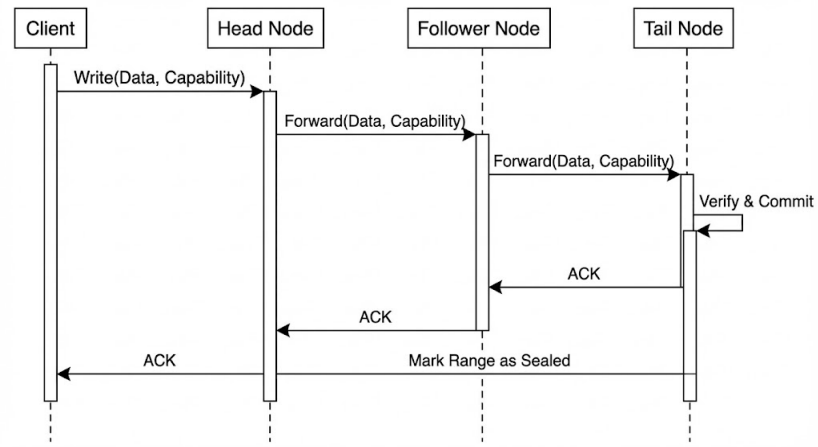

Figure 2: Seal-after-Commit Sequence. The Tail defines the commit point. Only after the Tail acknowledges the write do all replicas transition the affected byte range to a "Sealed" (immutable) state.

Storage Redundancy with DRBD. In deployments that want fast replacement after a node or disk failure without changing the chain replication protocol, each replica in the chain can be implemented as an active member with a hot standby whose local volume is mirrored via DRBD.

We utilize DRBD in synchronous Protocol C mode for this purpose.[41] The active node mounts the XFS filesystem and participates in the chain; the standby remains unmounted. Writes are acknowledged by the active node only after they are persisted on the standby. This provides durability against single-node hardware failures without engaging the global control plane.

The filesystem is the enforcement boundary for mutation. The LSM gate and SELinux policy operate on the mounted XFS tree and deny write-like system calls unless a valid capability has been presented and consumed. DRBD operates below the filesystem, mirroring the resulting blocks without participating in authorization decisions. Preventing user space from bypassing the gate by opening the raw device is essential.

When an active replica fails, the operations domain on the standby promotes DRBD to primary and mounts the protected filesystem. Because sealing metadata is persisted on the filesystem (for example via the seal journal described earlier), the standby inherits the sealed state upon mount. The standby then performs attestation renewal, obtains and installs a fresh boot ticket (advancing `boot_ctr` in a hardware-backed monotonic way), and only after that does the orchestrator publish an updated placement-group mapping and bump the placement-group epoch. Kernels reject capabilities minted under the old placement-group epoch and kernels on the newly promoted node reject sub-tokens minted for its previous boot counter, so replay attempts from the former topology fail.

3.2 Authorization and Token Bundles

Policy is expressed in an Operation Policy Manifest (OPM). For each scope of data, such as a project subtree, the OPM lists owner and operator keys, roles, approval thresholds, and constraints such as retention. When a client wants to perform a change it does not invent a token locally, instead, the head collects ephemeral holder public keys from each replica in the chain over a control channel. It also learns each replica's current boot counter as part of that replica's attested node identity. It then calls the authorizer with the operation type, the resource identifier, the placement group identifier and epoch, the chain identifier, and the set of ephemeral keys.

The authorizer verifies that the request matches policy and topology, increments a per-resource sequence number, and returns a token bundle. The bundle has a shared body and per-replica sub-tokens.[23] The body binds the operation, resource identifier, placement group identifier and epoch, chain identifier, and cryptographic algorithm parameters. For write-like operations it also binds the authorized byte interval and content metadata, so that both the kernel and the tail can enforce the same approved mutation window. For name-based operations it binds the path identifier and the expected current target file identifier (and, for replacement, the source file identifier), preventing substitution via rename. Each sub-token binds the operation to one replica by naming its node identifier and role, including its ephemeral key, and carrying the replica's boot counter `boot_ctr` as observed by the control plane. The authorizer signs both the body and each sub-token with its TEE-protected signing key. The head consumes its sub-token locally and forwards the remaining sub-tokens to followers. In this way, every replica has a capability specifically intended for that node and can verify that it is not replaying someone else's token and not replaying a token minted for a previous boot of the same node. Tokens are required for every write-like operation in the fully mediated profile, and at minimum for edits and deletes of sealed content and for administrative actions in the default profile.

3.3 Kernel Enforcement and Hardening

Enforcement is intentionally tied to the ordinary system call interface but hardened against the race conditions and granularity flaws that plague traditional userspace interposition. The storage daemon runs in a confined SELinux domain and performs I/O on behalf of clients. When it needs to perform an operation that requires a capability, it first invokes an authorization ioctl on the target file descriptor, or on a directory descriptor plus name for path operations. The ioctl carries the token bundle and a holder signature made with the ephemeral key known to that node.

The Granularity Advantage via eBPF. Standard Linux Security Modules (LSMs) typically authorize access based on file handles and masks (e.g., `MAY_WRITE`) but lack visibility into the specific byte offsets of a write operation. To overcome this limitation, our gate is implemented using eBPF. This allows the enforcement logic to attach directly to the kernel's `rw_verify_area` function, granting the gate visibility into the exact byte offsets $[off, off+len)$ of every write operation before any data is moved. The gate verifies the authorizer's signature on the capability body, checks that the sub-token's boot counter matches the kernel's active boot counter for the node, and validates the epoch alignment. Only if these deep checks pass does the gate install a *latch*.

Atomic Mutation Strategy. To eliminate the risk of torn writes or corruption during a crash, we strictly separate the *staging* of data from its *commitment*. For operations that modify existing sealed content (EDIT_RANGE), the storage daemon is not permitted to overwrite the live file directly. Instead, it must stage the new ciphertext into an anonymous temporary file (using `O_TMPFILE`). Once the payload is verified by the tail replica, the daemon invokes the `XFS_IOC_EXCHANGE_RANGE` ioctl. The kernel gate intercepts this specific ioctl, checks for a valid latch, and allows the filesystem to atomically swap the data extents of the live file with the temporary file. This ensures that the live file transitions from "valid old state" to "valid new state" instantly, with no window for partial data corruption.

Latch Semantics. If all checks pass, the gate installs a latch that is *task-scoped* and bound to the authorizing security context. For write-like operations, the latch stores the file identifier (or path identifier), the authorized byte interval, and a remaining budget. The latch is bound to the authorizing task (or thread group), the open file description being used, and the task's effective credentials and SELinux domain. This task-scoped design ensures that latches are not transferable via file-descriptor passing mechanisms: if a process receives a file descriptor via `SCM_RIGHTS` it does *not* gain any existing latch and must present its own capability to obtain a fresh latch.

New tasks created via `fork` or `clone` inherit latches only when they also inherit the same effective credentials and remain in the same SELinux domain; subsequent transitions that change credentials, such as `setresuid`, `setfsuid`, `execve` with setuid binaries, or entry into a different SELinux domain, invalidate any inherited latches. For name-based operations such as remove and replace, the gate installs a one-shot latch bound to the directory descriptor, name, and credentials and consumes it on the next matching namespace-changing syscall.

LSM Composition. The gate leverages the BPF-LSM framework (or compatible eBPF tracing hooks) to compose with SELinux under Linux's LSM stacking model.[1] SELinux reduces attack surface by controlling which domains can open protected files and invoke the authorization ioctl, while the eBPF gate enforces capability semantics across the covered write-like interfaces by inspecting arguments that are invisible to standard LSMs.

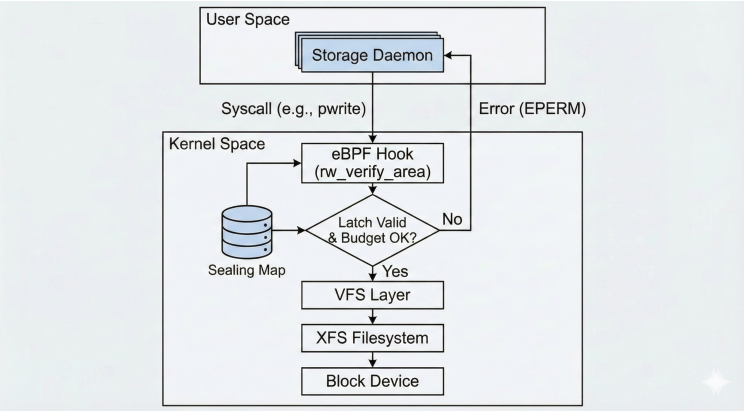

Figure 3: Kernel Enforcement Data Path. The Storage Daemon invokes syscalls. The eBPF Hook intercepts them at `rw_verify_area`, checking the Task-Scoped Latch. Valid operations pass to the VFS/XFS layer; invalid ones are rejected before modifying data.

Latch Scope and Memory Mappings. Real workloads write in multiple system calls, use vectored I/O, or rely on writable memory mappings. To support these patterns while keeping capabilities one-use, a capability may authorize a *range-scoped latch* that allows a bounded amount of write-like activity for a specific file identifier and byte interval. The latch is consumed when its budget has been exhausted, when the range has been fully written, or when the latch is invalidated by policy. Writable `mmap` and `mprotect` transitions are treated as write-like operations and require a latch; the budget is charged page-by-page as faults occur. However, for high-integrity mutations of sealed data, the atomic exchange path is preferred over memory-mapped I/O to guarantee crash consistency.

Host Hardening and SELinux. The SELinux policy on storage nodes is used to turn the host into an appliance that can be managed and observed but not arbitrarily rewritten, even by local root. Rather than granting a single privileged account broad powers, the policy divides responsibilities into confined domains.

The **Storage Daemon Domain** is the only component that may open files in the protected tree and call the authorization ioctl. It is explicitly denied access to raw block devices. The **Operations Domain** is responsible for DRBD promotion/demotion and mounting/unmounting the filesystem. It can open DRBD devices but is not allowed to open files under the protected tree. The **Authorizer Stub** domain runs the authorizer communication logic and talks to TEE/TPM devices but cannot access storage. Finally, the **Admin/Observability** domain allows operators to read logs and inspect status but prohibits them from mounting filesystems, changing policies, or signaling the storage daemons.

3.4 Resource Identity

The system needs stable, tamper-resistant identifiers for files and paths so that capabilities cannot be redirected. Each protected file stores a 16-byte identifier in the extended attribute `security.nlb.file_id`. Each protected directory stores a 16-byte identifier in `security.nlb.dir_id`. For a directory and a file name, the kernel computes a path identifier as

$$\mathsf{path\_id} = H(\text{“NLB-PATH”} \,\|\, \mathsf{dir\_id} \,\|\, \mathsf{utf8\_nfc}(\mathsf{name})).$$

The $\mathsf{utf8\_nfc}$ function applies Unicode normalization (NFC) to the pathname component before hashing. To avoid ambiguity on Linux filesystems that treat pathnames as opaque byte strings, protected trees enforce a naming invariant: each pathname component must be valid UTF-8 and already in NFC form on disk. Creates and renames in protected trees that would introduce a non-UTF-8 or non-NFC component are rejected. Under this invariant, $\mathsf{utf8\_nfc}$ is both well defined and injective over the namespace of protected trees, so the mapping from $(\mathsf{dir\_id}, \mathsf{name})$ to $\mathsf{path\_id}$ is unambiguous.

This restriction is deliberate and may reject legacy filenames that are not valid UTF-8. Within protected trees, such creates/renames fail (for example with `EINVAL` or a protocol-equivalent error). Deployments can scope protected trees to namespaces where this invariant is acceptable, while leaving other exports unmanaged for opaque-byte compatibility.

The gate uses file identifiers for file-descriptor operations and path identifiers for name-based operations. To prevent name-based capabilities from being redirected by swapping in a different inode at the same name, name-based operations additionally bind to the expected current

target file_id (when a target exists) and the kernel checks that the directory entry referenced by `(dirfd,name)` resolves to an inode whose `security.nlb.file_id` matches that expected value. Combined with the restrictions on hard links and cross-filesystem renames described below, this prevents user space from redirecting capabilities by manipulating directory entries.

Hard links to protected inodes are rejected and cross-filesystem renames inside protected trees are disallowed.[4, 20] This prevents user space from creating multiple names for the same inode or moving protected inodes into contexts where policy would not expect them.

Xattr lifecycle. The `security.nlb.file_id` and `security.nlb.dir_id` extended attributes are set exactly once by kernel-controlled paths. Directories receive their dir_id at creation. Files receive their file_id either at creation or, when created as unnamed temporary inodes (for example via `O_TMPFILE`), via a kernel gate operation that initializes a missing file_id exactly once as part of a successfully authorized WRITE_NEW. Subsequent attempts to modify these attributes from user space are denied. Optional EVM appraisal protects the integrity of the `security.*` xattrs so that tampering is detected and blocked under enforcing policy.[19] User space therefore cannot redirect capabilities by changing identifiers on disk and cannot retroactively modify identifiers on existing inodes.

3.5 Modification Paths and Atomic Primitives

A key design choice is to treat modification broadly. The gate enforces decisions on all paths that can change file content or layout, not only on simple writes.

This enforcement covers direct checks for file write permission, memory maps with write permission and `mprotect` transitions that add write to an existing map, and `ftruncate` or other attribute changes that alter file length. It also includes `fallocate` modes that punch, collapse, insert, or zero ranges, range copy and remap primitives such as `copy_file_range`, clone and deduplication controls provided by filesystems, and stream paths such as `splice`, `sendfile`, and related calls. Finally, io_uring write paths, which internally go through the same hooks, are also covered.

Crucially, the design explicitly supports and monitors the `XFS_IOC_EXCHANGE_RANGE` ioctl. This modern XFS primitive allows atomically swapping the data extents of two files. In our architecture, this is the *required* path for modifying sealed content: the writer stages data into a temporary file and, upon authorization, swaps it into the sealed file. The gate intercepts this ioctl, verifies that the target file has a valid active latch for the affected range, and permits the swap only if the source file (the temp file) matches the length and alignment constraints. In contrast, the legacy `XFS_IOC_SWAPEXT` and generic reflink deduplication interfaces are disabled or denied on sealed ranges to prevent unauthorized data manipulation.

Name-based operations that can introduce content, such as linking an `O_TMPFILE` into the namespace or using exchange renames, are also checked. We constrain sealed trees to buffered XFS mounts without DAX or hugetlbfs; direct access mappings that bypass the page cache and other filesystems with non-standard write paths are treated as out of scope for this design. An optional block-layer filter rejects raw writes to sealed logical block address ranges.[4, 21] Together, these checks aim to remove side doors around the main write paths so that sealed ranges cannot be modified through obscure interfaces.

The block-layer filter is treated as a secondary safety belt; the primary enforcement point is the filesystem-level gate, and correctness does not depend on block-level filtering. In particular, filesystems are free to issue TRIM or discard for blocks that previously belonged to sealed ranges and later reuse those blocks for unrelated data. When this behaviour would make it difficult to maintain a sound mapping from sealed file ranges to sealed logical blocks, operators can either disable discard for protected trees or disable the block-layer filter entirely. Integrity and immutability guarantees rely only on the gate; the block filter improves defence in depth when block allocation and reuse policies are well understood. Because XFS is used without reflink and with online defragmentation and extent-moving ioctls disabled for protected trees, the mapping from sealed file ranges to underlying logical block addresses remains stable whenever the block-layer filter is enabled.

Table 1: Coverage of write-like interfaces on protected XFS filesystems

Interface	Effect on sealed ranges	Enforcement
`XFS_IOC_EXCHANGE_RANGE`	**Atomic Content Swap**	**Primary Edit Path.** Requires latch; swaps temp data into sealed range.
`write`, `pwrite`	Direct content change	Denied on sealed ranges (use atomic swap instead)
`writev`, `pwritev`	Direct content change	Denied on sealed ranges
Writable `mmap`, `mprotect`	Memory-based change	Requires latch; discouraged for sealed data due to crash consistency risks
`ftruncate` / size change	Length / layout change	Denied when it would drop sealed bytes
`fallocate` modes	Range removal or shift	Denied on sealed extents
`copy_file_range`	Range copy/remap	Denied when source or dest intersects sealed extents
`FICLONE`	Reflink clone	Reflink disabled on protected mounts
`FIDEDUPERANGE`	Deduplication	Denied on sealed extents
`XFS_IOC_SWAPEXT`	Legacy Extent swap	**Denied** (superseded by EXCHANGE_RANGE)
`splice`, `sendfile`	Stream-based movement	Denied when destination is sealed range
io_uring write opcodes	Asynchronous writes	Routed through same hooks as `write`
Raw block I/O	Bypass filesystem	Denied by SELinux; optionally blocked by LBA filter

3.6 Filesystem State and Persistence

The default filesystem is XFS with reflink disabled. This choice is pragmatic. Sealing is defined over file data extents, not filesystem metadata blocks. The sealing map tracks byte ranges in terms of file identifiers and logical offsets, and, when the optional block-layer filter is enabled, in terms of the data extents that back those ranges. Metadata writes such as inode updates and allocation bitmap updates are not sealed, since they are required for correct filesystem operation and do not by themselves change committed ciphertext. Protection is opt-in at the path and mount level: operators label one or more XFS filesystems or directory subtrees as protected, and only those trees are subject to sealing and capability checks; paths outside those trees behave like a conventional Linux filesystem. Disabling reflink avoids hidden data movement in common maintenance paths, such as deduplication and snapshotting, and makes sealing simpler to reason about because blocks that back a sealed range are less likely to be reused for

other content.[4] Overlay and union filesystems are not supported for sealed trees because they introduce additional layers of indirection and hidden copy-on-write behavior. Deletion is treated as an authorized unlink at the namespace level. Once a name is removed, block reclamation and TRIM are left to the filesystem and device. The system does not attempt to provide secure erase and focuses on the guarantee that sealed data could not be modified before deletion.

Sealed-range metadata representation. To make sealing efficient and crash-tolerant, sealed ranges are tracked at a segment granularity tied to the encryption layout (as recorded in write metadata). The kernel maintains a per-mount seal map keyed by file_id and segment index, represented as a coalescing interval set (for example, an interval tree per file identifier). Lookups for IntersectsSealed are therefore $O(\log n)$ in the number of sealed intervals for that file, and adjacent sealed intervals are merged to reduce fragmentation. For persistence and recovery, the kernel appends MarkSealed records to a hidden, kernel-owned journal inode on the protected filesystem; on mount (or after crash recovery) the journal is replayed and compacted into the in-memory structure. Because sealing is idempotent and segment-aligned, replay is safe and compaction can be performed opportunistically under kernel control without changing the security semantics.

4 Placement Groups and Tail Routing

Placement groups are the unit of placement, ordering, and failure isolation. For each placement group, the orchestrator records an ordered list of replicas, defining a logical chain with a head and a tail.[22] Clients send ciphertext to the head. The head writes locally and forwards the same ciphertext, along with the capability metadata, to the next replica. Intermediate replicas do the same for operations that carry capabilities in the chosen profile: they authorize locally, persist the ciphertext, and forward it on.

The tail has a special role. It verifies the content commitment for the frame, persists the write, and only then sends an acknowledgement upstream. This acknowledgement is the commit point for that placement group. Once the head receives the acknowledgement from the tail, the write is considered committed. At that point, each replica marks the corresponding byte ranges as sealed.

This arrangement avoids introducing a separate routing proxy. Clients only need to know how to reach the storage service endpoint (for example, an SMB or NFS export). The storage daemon uses the placement-group mapping and epochs from the control plane and the capabilities it obtains from the authorizer to route mutations across tails. Consistent hashing of objects into placement groups spreads traffic across tails and yields load balancing without complicated control planes. Failures and quarantines are contained to the placement groups whose tails are affected. Other groups continue using their replicas and epochs unchanged.

5 Mandatory TEE Authorization and Client Integration

At the policy level the system supports two deployment profiles. The design deliberately distinguishes between *growth* (creation and append) and *alteration* (overwrite and delete) to balance security with performance.

In the **Strict Profile**, every write-like operation that changes bytes on disk is gated by a TEE-issued capability. This provides maximum assurance but incurs a latency penalty for every mutation transaction.

In the **Default Profile**, the system prioritizes high-throughput ingestion. Ordinary creation and growth (appending) of a user's own files rely on authenticated identity and filesystem permissions (the "Fast Path"), avoiding the TEE round-trip. However, edits to sealed ranges, overwrites, and deletions are treated as destructive actions and effectively require a TEE capability (the "mediated path"). In both profiles, tokens are short lived and one use. The integration is designed so that clients continue to use ordinary POSIX-facing protocols such as SMB3/CIFS and NFSv4 while the authorization protocol remains entirely server-side.[52–54]

Table 2: Operations requiring TEE-issued capabilities vs. Fast Path

Operation	Default profile	Strict profile
Growth (Fast Path)		
Create new file under allowed prefix	Identity + ACL (No TEE)	Capability required
Append / extend unsealed ranges	Identity + ACL (No TEE)	Capability required
Alteration (Mediated)		
Edit sealed byte range	Capability required	Capability required
Delete sealed object	Capability required	Capability required
Repair / catch-up	Capability required	Capability required
Placement-group rebalance	Capability required	Capability required
Node-level administrative actions	Capability required	Capability required

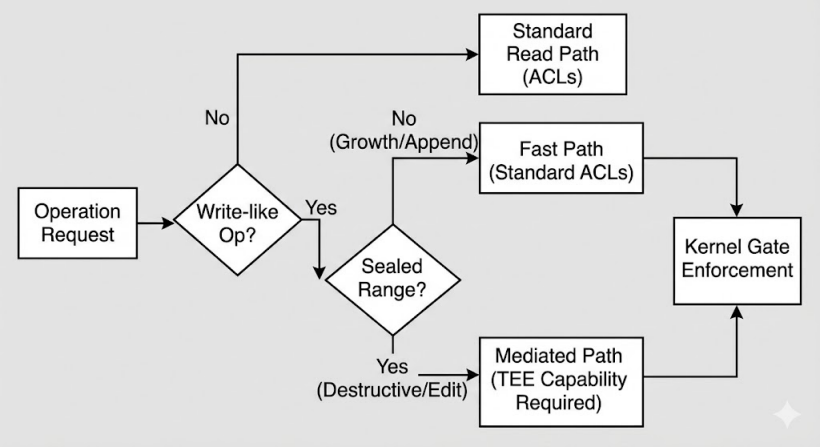

Figure 4: Authorization Decision Flow. Standard ingestion (Appends) follows the high-performance Fast Path (Green). Destructive mutations (Overwrites/Deletes) are routed to the TEE Authorizer (Red) for cryptographic validation.

This split ensures that write-heavy workloads such as logging, database appends, or backup ingestion operate at native speed, as they strictly append data. The performance cost of the TEE interaction is incurred only when a user (or attacker) attempts to modify data that has already been committed and sealed.

Client systems mount shares using standard protocol stacks (for example, SMB3 on Windows

or Linux, NFSv4 on Unix-like systems). The storage daemon on the head node terminates the client protocol, maps each request to a canonical operation on a specific file or path identifier, and, when policy demands (e.g., an attempt to overwrite a sealed range), obtains a capability bundle from the authorizer on the client's behalf.

Administrative roles are treated differently from ordinary users. For actions such as replacing existing content, removing objects, or performing catch-up operations, the authorizer demands step-up proofs from hardware authenticators such as FIDO2 tokens via Web Authentication or PIV smart cards, presented via the enterprise identity provider.[47–49, 51] Policy can require multi-party approval, for example one project administrator and one security officer must approve a removal. When the storage daemon requests a capability for such an operation, it includes a client identity token and any associated approval artefacts in its call to the authorizer. The authorizer verifies identity and attestation, checks policy, increments the sequence, and returns a capability bundle only when all conditions are satisfied. All approval traffic flows between clients, identity providers, and the authorizer; storage servers carry only the resulting capabilities and do not participate in the approval protocol.

Services obtain capabilities headlessly using mTLS identities such as SPIFFE IDs and may attach workload attestation from confidential VMs or enclaves to prove that the requesting process is running in an expected environment.[32, 33, 50] The storage daemon still mediates their filesystem operations and calls the authorizer on their behalf; the authorizer bases decisions on the service identity, attestation, and project policy recorded in the Operation Policy Manifest. Ordinary users operate on their own data under project policy. A common pattern is that creating new objects under project-controlled prefixes is allowed based on identity and ACLs alone, while replacements, removals, and range edits of sealed content require explicit capabilities. Visibility of objects is governed by UNIX permissions and optionally SELinux categories, so users cannot open paths outside their projects. Decrypt permission is governed separately by ABE headers and client attribute keys as described in the next section. Mutate and decrypt are distinct decisions and are handled by different services.

6 Client Protocol Integration and Canonical Operations

Clients access the system over ordinary storage protocols such as SMB3/CIFS and NFSv4.[52–54] The design does not require changes to client operating systems or libraries. Instead, a storage daemon on the head node terminates the client protocol, maps each request to a small set of protocol-independent operations—WRITE_NEW, EXTEND_APPEND, EDIT_RANGE, and REMOVE—and, when policy requires, calls the authorizer. The kernel gate enforces the resulting capabilities on the POSIX system calls that the storage daemon issues. From the client's perspective the mount behaves like a conventional share; capability issuance and enforcement are entirely server-side.

6.1 Mapping Protocol Requests to Canonical Operations

On SMB3 and NFSv4, the storage daemon receives requests that identify a session security context, a path or file handle, and a protocol opcode such as WRITE, SET_INFO, or REMOVE.[52–55] The daemon resolves the handle or path to a protected inode, derives a resource identifier (file_id from the `security.nlb.file_id` xattr or path_id from the parent directory identifier

and normalized name), reads the current file length, and classifies the request into one of the canonical operation types that the authorizer understands.

Table 3 sketches the mapping for common cases. The precise classification depends on the protocol opcode, whether the target inode already exists, and the relation between the requested byte interval and the current file length. The mapping is intentionally simple so that the storage daemon can implement it using the same metadata it already needs to serve SMB or NFS.

Table 3: Examples of mapping SMB/NFS requests to canonical operations

Protocol-level event	Local syscalls	Canonical operation
Create new file (no inode present)	`open(O_CREAT|O_EXCL)` + `pwrite`	WRITE_NEW on file_id
First write to empty file	`pwrite(fd, ..., off = 0)`	WRITE_NEW on file_id
Write strictly beyond current EOF	`pwrite(fd, ..., off` $\geq$ size)	EXTEND_APPEND on file_id
Write fully inside existing prefix	`pwrite(fd, ..., off+len` $\leq$ size)	EDIT_RANGE on file_id
Increase file length via truncate	`ftruncate(fd, new_len` $>$ size)	EXTEND_APPEND on file_id
Delete file (SMB delete-on-close, NFS REMOVE)	`unlinkat(dirfd, name)`	REMOVE on path_id

Whether a byte range is sealed is not part of this classification. The storage daemon always maps a request that rewrites bytes $[off, off+len)$ into an EDIT_RANGE with that interval; the kernel gate later decides whether such an edit is permitted in the current profile and sealed state. In particular, a client does not need a special network verb to "edit sealed range"; the semantics are determined by whether a matching latch exists when the daemon issues the underlying `pwrite(2)` or `ftruncate(2)`.

Repair, catch-up, and placement-group rebalance operations do not flow through client-facing protocols at all. They are separate control-plane RPCs driven by the orchestrator and authorizer and are handled by the storage daemons under those components' direction.

6.2 Client Identity and Authorization Requests

The storage daemon must authenticate client requests to the authorizer using client credentials, not unauthenticated local usernames. This is achieved by forwarding verifiable identity tokens obtained during protocol authentication.

For SMB3, the daemon uses Kerberos or NTLMv2-based authentication as specified in the SMB2/3 protocol.[52] The daemon obtains a security context for the client principal and, when Kerberos is in use, has access to a service ticket or PAC that is cryptographically bound to that principal. For NFSv4, the daemon uses RPCSEC_GSS to obtain a GSS-API security context that carries signed identity assertions for the client.[55–57] For custom or POSIX-over-TCP protocols, mutual TLS or OIDC access tokens provide similar signed identity information.[49]

The daemon first evaluates if the operation requires external authorization based on the active profile (see Table 2). For operations on the Fast Path (e.g., appends in the Default Profile), the daemon proceeds directly to filesystem ACL checks.

When an operation falls into the class that requires a capability (e.g., modifying a sealed range) and touches a protected tree, the head storage daemon constructs an authorization request to the authorizer. This request includes the canonical operation type (WRITE_NEW, EXTEND_APPEND, EDIT_RANGE, or REMOVE), the resource identifier (file_id or path_id) and, for write-like operations, the byte interval $[off, off+len)$. It also carries the placement-group identifier, chain identifier, and placement-group epoch derived from local metadata, as well as per-replica ephemeral holder public keys collected from the replicas in the chain over the control channel, along with each replica's current boot counter as asserted by its attested node identity. Finally, it includes a client identity token or ticket as received from the protocol stack (for example, a Kerberos service ticket or PAC, an RPCSEC_GSS credential, or an OIDC access token), along with any step-up proofs such as WebAuthn assertions when required.[47–49]

For name-based operations such as remove and replace, the storage daemon also includes the expected current file_id at that path when one exists. This expected target identifier is obtained by resolving the directory entry in-kernel and reading the file_id xattr of the referenced inode immediately before requesting authorization. Binding the capability to this identifier prevents a compromised daemon from authorizing deletion or replacement of *whatever happens to be at that name later*.

The authorizer validates the client token directly against the identity provider or KDC trust anchors and evaluates policy in the Operation Policy Manifest based on those claims. It does not rely on unauthenticated assertions from the head about the user's identity. If policy allows the operation, the authorizer allocates a fresh per-resource sequence number, constructs a capability body binding the operation, resource, epochs, and any content metadata (including the authorized byte interval), and attaches one sub-token per replica bound to that replica's node identifier, boot counter, and holder key. It signs the body and sub-tokens with its TEE-protected key and returns the resulting bundle to the head.

When clients use client-side encryption, they include content metadata such as the BLAKE3 content commitment, write salt, and segment layout in the same control-plane request that triggered capability issuance. The authorizer copies that metadata into the capability body so that the tail can recompute and verify the commitment during commit, as described earlier.

6.3 Presenting Capabilities and Executing Syscalls

After obtaining a capability bundle from the authorizer, the head storage daemon presents the local sub-token to the kernel gate via the authorization ioctl on its own file descriptor. The ioctl carries the capability bundle, the local replica sub-token, and a holder signature made with the ephemeral key that the node contributed when the bundle was minted. The kernel gate performs the same signature, boot counter, epoch, and per-resource sequence checks described earlier, derives the resource identifier from in-kernel state, and, if all checks succeed, installs a range-scoped latch bound to the authorized byte interval and a budget derived from the capability.[1] Because latches are task-scoped, a file descriptor alone does not confer mutation permission: the authorizing task must hold the matching latch at the moment it attempts the write-like syscall.

The head then forwards the write frame and the remaining sub-tokens to the followers in the chain. Each follower's storage daemon invokes the same authorization ioctl on its local file descriptor with its own sub-token. Followers do not contact the authorizer; they only present the sub-tokens minted for them. Once the ioctl succeeds, the storage daemon issues the same POSIX system calls it would use in a conventional stack, such as `pwrite(2)`, `ftruncate(2)`, or `unlinkat(2)`. As long as these calls touch only bytes inside the authorized range and stay within the budget, the kernel gate allows them to proceed. When the latch is exhausted or when a call would modify bytes outside the range, the gate denies the syscall and returns `EPERM`.

The tail replica recomputes the BLAKE3 content commitment from the frame header and ciphertext stream and compares it to the commitment in the capability body. Only if they match does it persist the frame, issue an `fdatasync(2)`, update the placement-group hash chain, and send an acknowledgement upstream.[36] Upon receiving the acknowledgement, all replicas mark the corresponding byte ranges as sealed as described earlier.

6.4 Error Handling and Client-visible Behavior

Authorization and enforcement failures are translated into conventional protocol error codes so that clients see familiar behaviour. The authorizer and kernel gate use specific errno values internally, but storage daemons map them to the closest SMB or NFS error statuses.

If the authorizer denies a request due to policy (for example, because the client principal is not allowed to delete a sealed object or a required approver has not signed off), the storage daemon does not attempt the underlying syscall. It returns a permission error to the client: a negative `EPERM` or `EACCES` for POSIX clients, an NFS NFS4ERR_ACCESS for NFSv4, or an SMB STATUS_ACCESS_DENIED for SMB3.[52, 53] To the user this appears as an ordinary "Permission denied" or "Access denied" on the mounted filesystem.

If the kernel gate rejects a capability during the authorization ioctl due to an epoch mismatch (`ESTALE`), a boot counter mismatch (`ESTALE`), or a non-increasing sequence number (`EALREADY`), the storage daemon treats this as a transient internal condition rather than as a client error. It refreshes placement-group epoch and node boot state from the orchestrator/authorizer, obtains a fresh capability bundle from the authorizer, and retries the operation once. Only if the condition persists does it return an I/O error such as `EIO` or a protocol-specific equivalent, such as NFS4ERR_IO or STATUS_IO_DEVICE_ERROR, to the client.[1, 52, 53]

If the kernel gate denies a write-like syscall because no latch is present, because the syscall would modify bytes outside the authorized interval, or because the budget has been exhausted, it returns `EPERM`. The storage daemon surfaces this as a permission failure indistinguishable from other server-side access control denials. In the strict profile, any attempt to change bytes on a protected filesystem without first presenting a valid capability therefore results in an immediate permission error at the client. In the default profile, unauthorized edits and deletes of sealed content behave the same way, while ordinary unsealed growth of a user's own files continues to be governed by identity and ACL checks alone.

In all cases, clients see only standard error codes and messages for their protocol. The additional machinery—per-resource sequence numbers, epochs, node boot counters, content commitments, and range-scoped latches—remains inside the authorizer and kernel gate and does not leak into client-visible APIs.

7 Keying Model with Attribute Based Encryption

Separation of roles. Encryption is always performed at the client. The tenant root key (TRK) is held either directly by the client or by a dedicated key service under tenant operator control. Storage nodes persist ciphertext only. They do not possess the TRK, they do not receive data-encryption keys (DEKs), and they do not hold ABE master secrets. The authorizer that issues mutation capabilities is separate from the key service. Its key material is limited to signing mutation tokens and cannot be used for decryption. Attribute issuance is performed by one or more ABE authorities that are not colocated with storage nodes.[34] The point of this split is that compromising a storage node or the authorizer alone is not enough to obtain plaintext. Recovering plaintext requires both (i) access to the TRK (or a permitted TRK-derived file key) and (ii) possession of ABE attribute keys that satisfy the header policy; compromise of one without the other is insufficient.

Hybrid encryption structure. Large objects are encrypted with an authenticated symmetric scheme at the client. HKDF is used in its standard RFC 5869 form with tenant-specific salts and domain-separated info strings.[35] For a given tenant root key (TRK) and file identifier file_id, the client derives a file key

$$K_f = \text{HKDF-Expand}\Big(\text{HKDF-Extract}(\mathsf{tenant\_salt}, \text{TRK}), \quad \text{“NLB-FILE”} \,\|\, \mathsf{file\_id} \,\|\, \mathsf{key\_version} \Big). \tag{1}$$

For each write, the client samples a fresh, uniformly random write salt write_salt and derives a data-encryption key

$$DEK = \text{HKDF-Expand}\Big(\text{HKDF-Extract}(\mathsf{write\_salt}, K_f), \quad \text{“NLB-DEK”} \Big). \tag{2}$$

The client encrypts content segments with an AEAD under DEK and computes a content commitment

$$H_{\mathsf{ct}} = \text{BLAKE3-256}(\mathsf{frame} \,\|\, \mathsf{ciphertext\_stream}),$$

with nonces derived by domain separation:

$$\mathsf{nonce} = \text{HKDF-Expand}\Big(\text{HKDF-Extract}(\mathsf{write\_salt}, \mathsf{file\_id}), \quad \text{“NLB-AEAD”} \,\|\, s \,\|\, \mathsf{cap\_id} \Big), \tag{3}$$

where s is the segment index and cap_id is the capability identifier for the operation. Each write uses a fresh write_salt and cap_id, and s is unique within that write, so nonces are unique under each DEK. These constructions rely on well studied primitives.[35–38] We treat HKDF as a PRF keyed by its input keying material and use fixed, human-readable info labels such as “NLB-FILE”, “NLB-DEK”, and “NLB-AEAD” to provide domain separation between different uses.

ABE-protected headers. To enable flexible decrypt authorization while keeping the TRK off storage nodes, the write salt write_salt and header parameters needed to derive the DEK are stored in a small header protected by ABE. To obtain ciphertext keys, a reader must satisfy the ABE policy *and* possess (or be able to obtain) the TRK for the relevant key_version.

Because many practical ABE schemes are CPA-secure and do not by themselves provide strong resistance to malleability, the header is structured as a small hybrid: the data needed to derive the DEK is protected under a symmetric AEAD key K_h sampled uniformly at random per write, and K_h is then encrypted under an ABE policy.[34] Concretely, the header plaintext contains write_salt and the segment layout parameters (and optionally key_version, aead_alg, and kdf_alg identifiers). The client computes

$$\mathsf{hdr\_ct} = \text{AEAD-Enc}(K_h,\ \mathsf{hdr\_plain},\ \mathsf{ad} = \text{“NLB-HDR”} \,\|\, \mathsf{file\_id} \,\|\, \mathsf{cap\_id}),$$

and stores $\mathsf{abe\_ct} = \text{ABE-Enc}(\mathsf{policy},\ K_h)$ alongside hdr_ct and the ciphertext stream. A reader that satisfies the header policy uses its ABE private key(s) to recover K_h, verifies and decrypts hdr_ct under the associated data, and then derives the DEK from TRK, file_id, key_version, and write_salt as above before decrypting the ciphertext locally.

ABE authorities and client keys. ABE authorities issue attribute keys to clients under enrollment rules that reflect organizational policy. Authorities hold the ABE master secret and are isolated from storage nodes. Clients hold ABE private keys bound to their attributes. A client that satisfies the policy in a header decrypts the ABE-encrypted header key, verifies and decrypts the header AEAD ciphertext, recovers write_salt and parameters, derives the DEK, and then decrypts ciphertext locally. Storage nodes cannot decrypt because they do not hold ABE private keys and never see the TRK.[34]

Revocation and rotation. Revocation is implemented using time-scoped or epoch-scoped attributes. When access should be reduced, operators publish a new epoch attribute and update policies so that future headers require the new attribute. New headers are produced under the updated policy, while bulk ciphertext remains unchanged. Clients without the current epoch attribute cannot decrypt new headers.[34]

Rotation of the tenant root key is supported by increasing the key version key_version recorded in the header and maintaining prior TRK versions for authorized reads. Writers use the current TRK and key_version; readers select the correct TRK version by the key_version in the header. Storage nodes are not involved in key rotation and never receive secrets.[34, 40]

Deletion and outage behavior. Deleting or stopping the authorizer halts mutation for operations that require capabilities because no new capabilities can be issued. Reads continue to return ciphertext and sealed ranges remain enforced. Deleting the ABE authority or its master secret prevents header decryption for future accesses and therefore blocks plaintext recovery, but does not affect integrity guarantees at storage. If the tenant root key is lost and there is no escrow, ciphertext becomes unreadable even for clients that satisfy ABE policies, because the DEK derivation depends on TRK.[39] None of these failure modes place decryption secrets on storage nodes.

Summary of guarantees. In summary, storage nodes never store or receive the tenant root key, data-encryption keys, ABE master secrets, or client attribute keys. They persist ciphertext and ABE-protected headers only. Permission to change stored data is independent from permission to decrypt it. Clients that hold valid attributes and can obtain the appropriate

TRK version can decrypt headers and then decrypt ciphertext locally. Storage nodes cannot produce cleartext even under full user-space compromise.[34]

8 Operations, Token Schema, and Algorithms

This section describes what information capabilities carry and how different operations constrain them.

All capabilities share a common body. The body includes a version, a capability identifier, a per-resource sequence number, a resource identifier (file or path), a global epoch, a placement-group identifier, chain identifier and placement-group epoch, a hash of the current Operation Policy Manifest, AEAD and KDF algorithm identifiers, and a key version. For write-like operations the body also carries the authorized byte interval $[off, off+len)$ and content metadata (content commitment, write salt, and segment layout) so that the tail and kernel enforce the same bounded window of mutation. For name-based operations the body carries the path identifier and the expected current target file identifier (and, for replacement, the source file identifier), preventing substitution via rename between authorization and execution. The bundle also contains one sub-token per replica. Each sub-token names the node identifier and role, includes the node's current boot counter boot_ctr, and includes the replica's ephemeral key. The authorizer signs both the body and each sub-token so that kernels can verify that a given sub-token is intended for the local node, matches the approved operation, and is not replayed across node reboots.[1, 23]

Capabilities are intentionally short lived in a logical sense rather than in wall-clock time. We do not rely on node clocks or timestamps in the token body. Instead, a capability becomes unusable as soon as its cap_seq has been consumed by the kernel gate for the bound resource, when the global or placement-group epoch changes, or when the node boot counter changes. This avoids creating correctness dependencies on clock synchronization while still bounding the lifetime and replay window of each capability. When operators need to revoke outstanding, unused capabilities due to policy change (for example emergency operator removal or enabling a legal hold), they advance the global epoch as described in the threat model; kernels then deterministically reject previously issued capabilities without requiring timestamps.

Different operations bind different metadata. For **Write new file**, the token carries the content commitment, write salt, segment size and count, total byte count, an external key identifier, and the authorized interval $[0, total\_bytes)$. The new file identifier is either bound directly as the resource identifier and initialized by the gate exactly once on the destination inode, or is verified against an already initialized file_id. For **Replace**, the token binds to a path identifier and records the source file identifier that will replace the target and, when the target exists, the expected current target file identifier. The kernel checks that the directory entry resolves to an inode whose file_id matches the expected target identifier and that the temporary inode used as the source has the expected source identifier, preventing substitution. For **Remove**, the token binds to a path identifier and records the expected current target file identifier. The kernel checks that the directory entry being unlinked resolves to an inode whose file_id matches that expected value. For **Edit range**, the token binds a file identifier, an explicit byte interval $[off, off+len)$, and the content commitment of the replacement stream. This allows the tail to verify that the bytes written match the approved content and allows the gate to install a range-scoped latch for that interval. For **Extend/append**, the token

binds the authorized append interval, the first new segment index, the number of segments, an aggregate commitment, the write salt, and the new byte count. Finally, specialized **Repair and catch-up** tokens allow a donor replica to expose sealed ciphertext ranges for copying and authorize the new node to accept those ranges as committed. These operations are restricted to ranges that are already sealed on at least one healthy replica.[4, 22]

Concurrency on a resource. Capabilities carry a strictly increasing per-resource sequence number. The authorizer serializes issuance per resource: when two clients concurrently request conflicting mutations on the same file or path, the authorizer either denies one request according to policy or issues capabilities with distinct sequence numbers. Replicas apply committed operations in the order in which the tail acknowledges them. Because the kernel gate enforces a monotonic cap_seq fence, storage daemons must submit operations for a given resource in increasing cap_seq order on each replica; if an implementation reorders operations for the same resource, later sequence numbers can be accepted first and earlier ones will be rejected and must be reissued. This policy yields deterministic resolution under tail commit order, while allowing a rebooted node with reset in-kernel state to accept newer cap_seq values (boot fencing prevents replay of older sub-tokens across that reboot).

8.1 Core Algorithms

The following algorithms detail the strict, fully mediated profile. In the default profile the same mechanisms apply only to edits and deletes of sealed content, while creation and unconstrained growth use the fast path.

Algorithm 1 Type Sketch & Token Schema

type FileId = bytes16; **type** DirId = bytes16; **type** PathId = bytes32
type Epoch = uint64; **type** CapId = bytes16; **type** Hash32 = bytes32
type BootCtr = uint64
enum Op ∈ { WRITE_NEW, REPLACE, REMOVE, EDIT_RANGE, EXTEND_APPEND, RANGE_COPY }
record Range {off : $u64$; len : $u64$}
record WriteMeta {H : Hash32; $write\_salt$: $bytes32$; seg_size : $u32$; seg_count : $u64$; $total\_bytes$: $u64$; key_id : $u32$; $range$: Range}
record ReplaceMeta {src_file_id : FileId; dst_file_id? : FileId}
record RemoveMeta {dst_file_id : FileId}
record TokenBody {$capid$: CapId; cap_seq : $u64$; op : Op; $resource$: (FileId | PathId); $epoch$: Epoch; pg_id : $u64$; $chain\_id$: $u64$; pg_epoch : $u64$; opm_hash : Hash32; $aead\_alg$: $u16$; kdf_alg : $u16$; $key\_version$: $u32$; $write$? : WriteMeta; $replace$? : ReplaceMeta; $remove$? : RemoveMeta}
record ReplicaProof {node_id : $u128$; $role$: $u8$; $boot\_ctr$: BootCtr; K_e^{pub} : bytes; rp_sig : bytes}

Algorithm 2 TEE_Evaluate_Policy($op, resource, epoch, approver\_keys, K\_e^{pub}$)

$opm \leftarrow$ FetchAndVerifyOPM($resource$)
$P \leftarrow opm.ops[op]$
$cap\_seq \leftarrow$ NextSeq($resource$)
$session \leftarrow$ Rand(); $ctx \leftarrow H(\text{“NLB-CTX”} \,\|\, op \,\|\, resource \,\|\, epoch \,\|\, cap\_seq \,\|\, session \,\|\, K\_e^{pub})$
Collect proof of possession signatures and verify against P
if count_valid $\geq$ threshold(P) **then**
    return $\langle \mathsf{ALLOW}, opm\_hash, cap\_seq, session \rangle$
else
    return $\langle \mathsf{DENY}, \mathsf{reason} \rangle$
end if

Algorithm 3 TEE_Issue_Multi($op, resource, epoch, pg, chain, \{(\mathsf{node\_id}, \mathsf{boot\_ctr}, K_e^{pub})\}, write?, replace?, remove?$)

$\langle verdict, opm\_hash, cap\_seq, session \rangle \leftarrow$ TEE_Evaluate_Policy($\cdot$); **assert** $verdict =$ ALLOW
$capid \leftarrow$ uuid4()
$body \leftarrow \langle capid, cap\_seq, op, resource, epoch, pg.id, chain.id, pg.epoch, opm\_hash, aead\_alg, kdf\_alg, key\_version, write?, replace?, remove? \rangle$
For each replica $(\mathsf{node\_id}_i, \mathsf{boot\_ctr}_i, K\_e^{pub(i)})$ in chain set
    $rp\_msg_i \leftarrow$ encode($\mathsf{cap\_id}, cap\_seq, op, resource, epoch, pg.id, chain.id, pg.epoch, \mathsf{node\_id}_i, role_i, \mathsf{boot\_ctr}_i, K\_e^{pub(i)}$)
    $rp_i \leftarrow \langle \mathsf{node\_id}_i, role_i, \mathsf{boot\_ctr}_i, K\_e^{pub(i)}, \text{Sign}_{K_{\mathsf{tee}}}(rp\_msg_i) \rangle$
$body\_sig \leftarrow \text{Sign}_{K_{\mathsf{tee}}}(\text{encode}(body))$
return $bundle \leftarrow \langle body, \{rp_i\}, tee\_key\_id, issuer\_id, body\_sig \rangle$

Algorithm 4 Client_Present_And_Execute

Head collects $(\mathsf{boot\_ctr}_i,\ K\_e^{pub(i)})$ and attested node identity from each follower over the control channel
$bundle \leftarrow$ TEE_Issue_Multi($op, resource, epoch, pg, chain, \{(\mathsf{node\_id}, \mathsf{boot\_ctr}, K\_e^{pub})\}, write?, replace?, remove?$)
Head signs the holder message with $K\_e^{(\mathsf{node\_id}_0)}$ and invokes Ioctl_Authorize
Head forwards the sub-token for each follower. Followers sign with $K\_e^{(\mathsf{node\_id}_i)}$ and invoke Ioctl_Authorize

Algorithm 5 Gate_Authorize(fd or $(dirfd, name), op, epoch, bundle, sig\_e$)

Parse $bundle = \langle body, \{rp_i\}, \ldots, body\_sig \rangle$
verify VERIFYSIG(K_{tee}, ENCODE($body$), $body\_sig$)
Locate local rp; **verify** VERIFYSIG(K_{tee}, ENCODE($rp \setminus rp.\mathsf{rp\_sig}$), $rp.\mathsf{rp\_sig}$) and match $rp.\mathsf{node\_id}$ and $rp.\mathsf{role}$
check boot counter $rp.\mathsf{boot\_ctr} = \mathsf{ActiveBootCtr}(rp.\mathsf{node\_id})$ **else return** ESTALE
verify holder VERIFYSIG($rp.K\_e^{pub}$, $\mathsf{encode}(\mathsf{holder\_msg}(body))$, sig_e)
check global epoch $body.epoch = \mathsf{ActiveGlobalEpoch}()$ **else return** ESTALE
check placement-group epoch $body.pg\_epoch = \mathsf{ActivePgEpoch}(body.pg\_id)$ **else return** ESTALE
check sequence $body.cap\_seq > \mathsf{LastSeenSeq}(body.resource)$ **else return** EALREADY
if $op \in \{\mathsf{WRITE\_NEW}, \mathsf{EDIT\_RANGE}, \mathsf{EXTEND\_APPEND}\}$ **then**
    $inode \leftarrow$ INODEFROMFD(fd); $f\_id \leftarrow$ GETXATTR($inode$, `security.nlb.file_id`)
    if $f\_id = \varnothing$ **and** $op = \mathsf{WRITE\_NEW}$ **then**
        INITXATTRONCE($inode$, `security.nlb.file_id`, $body.resource$) ▷ set-once kernel path
    end if
    assert GETXATTR($inode$, `security.nlb.file_id`) $= body.resource$
else if $op = \mathsf{REPLACE}$ **then**
    $d\_id \leftarrow$ GETXATTR(INODEFROMFD($dirfd$), `security.nlb.dir_id`)
    assert $H(\text{“NLB-PATH”} \,\|\, d\_id \,\|\, \mathsf{utf8\_nfc}(\mathsf{name})) = body.resource$
    $dst \leftarrow$ LOOKUPDENTRYINDIR($dirfd, name$)
    if $body.replace.dst\_file\_id$ is present **then**
        assert $dst \neq \varnothing$ and GETXATTR($dst.inode$, `security.nlb.file_id`) $= body.replace.dst\_file\_id$
    else
        assert $dst = \varnothing$ ▷ replacement into a non-existent name
    end if
    assert src_id on the temp inode matches $body.replace.src\_file\_id$
else if $op = \mathsf{REMOVE}$ **then**
    $d\_id \leftarrow$ GETXATTR(INODEFROMFD($dirfd$), `security.nlb.dir_id`)
    assert $H(\text{“NLB-PATH”} \,\|\, d\_id \,\|\, \mathsf{utf8\_nfc}(\mathsf{name})) = body.resource$
    $dst \leftarrow$ LOOKUPDENTRYINDIR($dirfd, name$)
    assert $dst \neq \varnothing$ and GETXATTR($dst.inode$, `security.nlb.file_id`) $= body.remove.dst\_file\_id$
end if
SETLASTSEENSEQ(body.resource,body.cap_seq)
Install a task-scoped latch bound to the open file description (or dirfd/name) and the current effective credentials and SELinux domain.
if $op \in \{\mathsf{WRITE\_NEW}, \mathsf{EDIT\_RANGE}, \mathsf{EXTEND\_APPEND}\}$ **then**
    Store authorized interval $body.write.range$ and a budget derived from $body.write.range.len$; consume latch when budget exhausted or on next conflicting syscall.
else
    Store a one-shot operation latch; consume on the next matching namespace syscall.
end if
return OK

Algorithm 6 Enforce_Write_Hook($fd, op, [off, off{+}len)$)

$sealed \leftarrow$ IntersectsSealed(file_id(fd), $[off, off{+}len)$)
$L \leftarrow$ LookupLatch($fd, op,$ Creds())
if StrictProfile() **or** *sealed* **then**
    if $L \neq \varnothing$ **and** $[off, off{+}len) \subseteq L$.range **then**
        ConsumeOrDecrementLatch(L, len); **return** ALLOW
    else
        return DENY
    end if
else
    return ALLOW ▷ Default profile fast path
end if

Algorithm 7 Replace_Path($dirfd, name, new\_file\_id$)

$resource \leftarrow H$("NLB-PATH" ‖ GetXattr($dirfd$, `security.nlb.dir_id`) ‖ utf8_nfc(name))
$dst \leftarrow$ LookupDentryInDir($dirfd, name$)
if $dst \neq \varnothing$ **then**
    $dst\_id \leftarrow$ GetXattr($dst.inode$, `security.nlb.file_id`)
else
    $dst\_id \leftarrow \varnothing$
end if
$replace \leftarrow$ {src_file_id $= new\_file\_id$, dst_file_id? $= dst\_id$}
$bundle \leftarrow$ TEE_Issue_Multi(REPLACE, $resource, epoch, pg, chain,$ {(node_id, boot_ctr, K_e^{pub})}, $\varnothing, replace, \varnothing$)
Client_Present_And_Execute(bundle)
Invoke `renameat2`. The gate checks destination path binding, expected target identifier (if present), and the source identifier

Algorithm 8 Remove_Path($dirfd, name$)

$resource \leftarrow H$("NLB-PATH" ‖ GetXattr($dirfd$, `security.nlb.dir_id`) ‖ utf8_nfc(name))
$dst \leftarrow$ LookupDentryInDir($dirfd, name$); **assert** $dst \neq \varnothing$
$dst\_id \leftarrow$ GetXattr($dst.inode$, `security.nlb.file_id`)
$remove \leftarrow$ {dst_file_id $= dst\_id$}
$bundle \leftarrow$ TEE_Issue_Multi(REMOVE, $resource, epoch, pg, chain,$ {(node_id, boot_ctr, K_e^{pub})}, $\varnothing, \varnothing, remove$)
Client_Present_And_Execute(bundle)
Call `unlinkat`. The gate checks path binding and expected target identifier; the latch allows one bounded attempt

Algorithm 9 Edit_Range_Atomic($fd, [\ell, L), new\_cipher\_stream$)

$(H, \mathsf{write\_salt}, total\_bytes, seg\_count, seg\_size, key\_id) \leftarrow$ HashFrame(new_cipher_stream)
$write \leftarrow \{H, \mathsf{write\_salt}, seg\_size, seg\_count, total\_bytes, key\_id, \{\mathsf{off} = \ell, \mathsf{len} = total\_bytes\}\}$
$bundle \leftarrow$ TEE_Issue_Multi($\mathsf{EDIT\_RANGE}, \mathsf{file\_id}(fd), epoch, pg, chain, \{(\mathsf{node\_id}, \mathsf{boot\_ctr}, K\_e^{pub})\}, write, \varnothing, \varnothing$)
Client_Present(bundle)
▷ Staging Phase: Write ciphertext to a hidden temp file first
$tmp\_fd \leftarrow$ open($O\_TMPFILE$)
pwrite($tmp\_fd, \dots$)
Chain_Forward($bundle, new\_cipher\_stream$)
▷ Commit Phase: Only swap data into the live file after Tail confirms
wait for Tail_Accept
$args \leftarrow \{file1\_fd : fd,\ file2\_fd : tmp\_fd,\ flags : \mathsf{XFS\_EXCHANGE\_RANGE\_DSYNC}\}$
ioctl($fd, \mathsf{XFS\_IOC\_EXCHANGE\_RANGE}, args$)
close(tmp_fd)

Algorithm 10 Bump_Epoch(pg_id, new_epoch)

Set the active placement-group epoch to new_epoch for the placement group. Replicas apply before accepting new operations
Advance sequence fences for resources in the group

Algorithm 11 Audit_Issuer_Kernel

Issuer logs capability identifiers,sequences,time of issue,operation,resource,epochs,manifest hash,approvers,authorizer key identifier,issuer identifier,node boot counters when applicable
Kernel logs attempts or successes with timestamps,capability identifiers,sequences,operation, resource,device and inode references or directory inodes,ranges and byte counts when applicable,result codes,authorizer key identifiers,boot counter checks

Algorithm 12 Chain_Write($pg, chain, op, resource, meta$)

The head obtains and presents the bundle locally,streams ciphertext to storage,and forwards the stream to the next node
for $i = 1$ to t **do**
    Replica $\mathsf{node\_id}_i$ authorizes with its sub-token,persists,and forwards if not the tail
end for
The tail acknowledges to the head. Replicas seal the committed prefix

Algorithm 13 Catch_Up(pg, new_node)

Increase the group epoch and publish a chain with the new node as tail
Issue range copy permission to donors and corresponding write permission to the new node for committed ciphertext
Copy sealed ranges until equal,mark them sealed,and switch the tail role to the new node

Algorithm 14 Heal / Rebalance

1: On failure select a replacement,bump the epoch,run catch up,and retire the failed node once safe
2: For planned redistribution move a set of placement groups with the same range copy procedure while epochs fence old chains

9 Authorizer Deployment, High Availability, and Recovery

The authorizer and orchestrator form a small control plane that is logically separate from the storage data plane. In a practical deployment this control plane runs on a set of non-storage nodes and exposes a single logical endpoint to the cluster, even when multiple replicas are present. The same mechanism that provides high availability and capacity for capability issuance also determines how the system behaves when the authorizer is lost or must be rotated.

9.1 Control Plane Topology and High Availability

The control plane consists of one or more authorizer replicas, each running on a control-plane node. Every replica hosts an authorizer process in normal world and a TEE instance that holds the authorizer signing key. All replicas participate in the same logical authorizer domain: they use the same signing key material inside their TEEs and connect to a shared control-plane store for policy, placement-group state, per-node boot counters, and per-resource `cap_seq` counters. From the perspective of storage nodes there is a single logical authorizer, even though multiple replicas may be available.

Storage nodes never talk to the TEE directly. They contact the authorizer over a mutually authenticated transport such as mTLS, e.g. at a fixed endpoint `authorizer.example.com`. The authorizer process terminates TLS, authenticates the caller based on its client certificate or SPIFFE identity, performs policy checks using the Operation Policy Manifest, allocates a fresh capability sequence number for the target resource from the control-plane store, constructs a canonical capability body, hashes it, and sends only the digest and minimal metadata into the TEE for signing. The TEE returns an Ed25519 signature under the shared authorizer key. The signed body includes an `issuer_id` field that identifies which replica issued the capability and, optionally, a measurement or enclave identifier. Because this identifier is part of the signed payload it is stable for honest replicas and useful for tracing and debugging. However, signatures from any replica are verified against the same authorizer public key in kernel keyrings, so all replicas are equally trusted within a given epoch.

Consistency for capability issuance comes from a shared control-plane store rather than from the TEEs themselves. A small replicated database or consensus-backed key-value store maintains per-resource `cap_seq` counters, per-node boot counters, Operation Policy Manifests and their hashes, placement-group mappings and epochs, and the latest hash-chain checkpoints for each placement group. Authorizer replicas issue capabilities concurrently but serialize on this store when allocating the next sequence number, reading each node's current `boot_ctr`, or updating the placement-group mapping, so that no two replicas emit capabilities with the same `cap_seq` for one resource. In practice, the control-plane store runs as a conventional $(2f+1)$-node cluster that tolerates f failures and requires a majority quorum for writes. The nodes that host the authorizer replicas may also host members of the store, or the store can run on a separate set of

control-plane nodes. Authorizer replicas treat the store as the single source of truth and refuse to issue capabilities if they cannot reach a quorum, failing closed for mutations while leaving reads and sealed-state enforcement intact on storage nodes.

9.2 Authorizer Loss and Recovery

It is important that losing the authorizer control plane does not force an emergency on the data side. In this design, individual replica failures are handled by the load-balancing and failover mechanisms described above; as long as at least one healthy replica and a control-plane quorum remain, capability issuance continues normally. The interesting failure modes are loss of a majority of the control-plane store, loss of all authorizer replicas, or compromise of the authorizer signing key.

Replica loss. Failure of a single authorizer replica, such as a process crash or node outage on one control-plane node, is handled transparently. Health checks on the virtual endpoint remove the failed replica from rotation and shift traffic to remaining replicas. Because replicas are stateless apart from the shared store, no special recovery is required on the storage side. Storage nodes continue to obtain capabilities from the same logical endpoint, and the kernel gate keeps enforcing sealing according to the active epochs, boot counters, and sequence numbers.

Loss of quorum or total authorizer outage. If the control-plane store loses quorum or all authorizer replicas become unavailable, mutation authorizations for operations that require capabilities simply cease. Writes, deletes, healing operations, and rebalances that require fresh capabilities do not proceed. Already sealed data remains enforced. Reads continue to return ciphertext, and existing replicas keep serving committed data. Replication of already committed data can continue when it does not require new authorizations. From the perspective of the data plane, the system becomes read only and immutable for sealed content until a healthy authorizer cluster with a functioning store returns.[1, 2]

Recovery in this case has three steps. First, operators restore authorizer replicas and the control-plane store to a healthy state, either on the same control-plane nodes or on replacements. Second, they rotate the authorizer signing key if compromise is suspected and distribute the new public key material into kernel trusted keyrings via the same signed update path used during initial deployment.[17, 18] Third, they bump the placement-group or global epoch so that kernels accept a new sequence domain and reject replays from the original domain. Once the new epoch is active and a quorum of the control-plane store is available, issuance resumes and storage nodes begin accepting capabilities from the restored authorizer cluster.

Loss of authorizer state. Loss or corruption of state in the control-plane store, such as per-resource counters, per-node boot counters, or placement-group mappings, is also handled by the epoch fence. When operators rebuild or restore the store, they promote a fresh view of policy and placement-group mappings and bump the relevant epochs. Kernels reject non-increasing sequence numbers from the old domain and accept fresh sequences under the new epochs. Attempted substitution of a fake authorizer is defeated by the kernel's trust anchors: kernels only trust pre-installed public keys and associated measurements. With lockdown enabled, a compromised root account on a storage node cannot add new trust anchors to the kernel keyring or replace existing ones.[18]

The same mechanism applies when a control-plane node is rebuilt or reintroduced after a reboot: the orchestrator increments the placement-group epoch before replicas resume service, so capabilities issued under a previous epoch cannot be replayed against a fresh kernel even if its in-memory cap_seq state was lost. Boot fencing on storage nodes ensures that after reboot they do not accept capabilities for their placement groups until they have observed the current global epoch and placement-group epochs from the restored control plane and have installed their current boot ticket, which advances the per-node boot_ctr in a hardware-backed monotonic way and makes prior sub-tokens for that node invalid.

The net effect is that authorizer replica failures are absorbed by the control-plane cluster, and only loss of quorum or deliberate rotation away from a compromised authorizer domain leads to a temporary loss of liveness for sensitive mutations. In all of these cases integrity and confidentiality at the storage layer remain intact: sealed ranges stay enforced, ciphertext remains unreadable on storage nodes, and the system fails closed for operations that require explicit authorization.

10 Commit Hash Chain and Drift Detection

Replicas may diverge due to hardware faults or malicious behavior even if authorization succeeds. To detect this, the system maintains a compact hash chain per placement group.

For each placement group, the tail and followers maintain a chain head H_n updated for every committed operation:

$$H_n = \text{BLAKE3-256}(H_{n-1} \,\|\, \mathsf{capid} \,\|\, \mathsf{resource} \,\|\, \mathsf{bytes} \,\|\, H_{\mathsf{ct}}).$$

Because commits are totally ordered, honest replicas will produce identical chain heads for the same index n.

Verification is asynchronous but deterministic. The tail periodically broadcasts a signed checkpoint containing the current index and chain head. Followers compare this against their local state *only* once they have applied commits up to the checkpoint's index. This strictly ordered comparison effectively distinguishes network lag from storage divergence. A mismatch at the same index indicates corruption; if a retry (to rule out transient errors) confirms the mismatch, the node is flagged for quarantine. In two-replica deployments, this signal can trigger a block-level DRBD online verification to isolate the faulty node.[36, 45]

11 Compromise Detection and Quarantine

The system treats certain signals as evidence that a node should no longer participate in the data path. These include replica divergence, failure of attestation, and inconsistencies between authorizer and kernel audit logs. Detection and quarantine are driven from outside the suspect node and do not rely on cooperation from compromised processes.

Detection combines three sources. First, the hash chain per placement group, described earlier, detects when a replica's committed history differs from the tail's at the same index.[36] Second, DRBD's online verification computes checksums over ranges of blocks and compares them between peers, catching divergence at the block level (for example within an active/standby

mirror pair or in two-data-replica deployments).[45] Third, attestation and audit mechanisms track trustworthiness. Node certificates used for replication and control are short lived and require a fresh TEE or TPM quote at renewal time. Each successful renewal advances the node's boot counter boot_ctr in the control plane, and kernels accept sub-tokens only for the active boot_ctr. Failure to renew or a quote that does not match the allow list is treated as a loss of trust.[32, 33] The authorizer logs all issued capabilities. Kernels log all consumed capabilities. A persistent discrepancy between issued and consumed for a node, such as capabilities that are consumed with unexpected operations or resources, is a strong signal of malfunction or compromise.

Quarantine starts at the authorizer. The control plane marks the node as quarantined and revokes its node certificate. The orchestrator increments the epochs for placement groups that included that node. Kernels reject capabilities with old epochs, and the authorizer stops minting sub-tokens for the quarantined node. In effect, the node is no longer part of any replica chain.

Peers then disconnect DRBD connections to the quarantined node and block replication and control traffic to and from its address using packet filters. If the node remains reachable, operators can demote its DRBD resources, unmount the protected filesystem, and set the block devices read only. SELinux remains in enforcing mode and policy already prevents the storage daemon from opening block devices, so even a compromised storage daemon cannot write to raw devices.

The rest of the cluster continues operating as long as quorum remains. In a two data plus voter or three data replicas layout, quarantining one node still leaves a majority of votes and at least one up-to-date data replica. Mutations for affected placement groups are directed to the remaining replicas according to the updated chain and epoch. When a quarantined node is repaired or rebuilt, it must produce a fresh attestation quote that matches the allow list. A new node certificate is issued, the node's boot_ctr advances, and (when DRBD standby mirroring is used) DRBD attaches it as a secondary. The node discards its old contents and resynchronizes from an authoritative peer. After resynchronization and successful online verification, the node can rejoin placement groups.

The authorizer exposes minimal interfaces to support quarantine and epoch management, specifically a quarantine call for nodes and an epoch-bump call for placement groups. These operations are idempotent and auditable and allow either operators or higher-level automation to drive quarantine and recovery without granting them direct access to raw devices or sealed data.

12 Related Work and Positioning

This work sits at the intersection of storage, operating systems, and applied cryptography. It uses ideas from several areas but combines them in a way aimed at protecting data under server compromise.

Trusted execution environments (TEEs) provide isolated storage and computation intended to resist a compromised host. OP-TEE secure storage, for example, offers a per-device persistent store for trusted applications, supporting REE filesystem and RPMB backends.[23] It provides confidentiality and integrity for small state and keys on one device. It does not address distributed

replication, chain replication, or immutability of large datasets under host-level compromise of general processes. In this design, TEE storage is used to protect the authorizer's signing key and node identities, but data immutability remains enforced by the kernel on sealed ciphertext.

Linux offers read-time verification through fs-verity and dm-verity, which authenticate file or block device contents against a Merkle root and refuse reads that do not match.[3, 24] These mechanisms are powerful but focus on detecting corruption when data is read. They do not control who may write new bytes into files before they are sealed and do not coordinate across replicas or enforce one-shot mutation tokens. Sealing and the kernel gate fill that gap by controlling mutation at the time it is proposed and by binding mutations to a TEE-issued capability. Verity can be layered on top when read-time integrity proofs are required, but it is not a substitute for kernel-enforced immutability of committed ciphertext.

Object-lock and WORM controls in cloud object stores provide retention windows during which objects cannot be deleted or overwritten by ordinary users.[25] These are valuable for compliance and records retention. However, they are typically implemented at the service API layer and are not bound to kernel identities or to all on-host write paths. The design here instead focuses on the general-purpose Linux host and interposes at kernel hooks for all write-like interfaces, binding authorization to identities derived inside the kernel.

Local snapshot systems such as those in ZFS and Btrfs provide point-in-time copies and policy-based retention.[26] Snapshots are excellent for recovery, but they do not prevent a process from encrypting new data or punching holes in ranges not yet captured by a snapshot, and a sufficiently privileged operator can still destroy snapshots altogether. Sealing aims instead to block those mutations at the system-call boundary so that once a range is sealed, that ciphertext remains stable until deletion, independent of snapshot policy. Snapshots can then be used to reduce recovery time objectives on top of that immutability.

Block-replication tools such as DRBD and controllers such as LINSTOR provide synchronous replicas and resynchronization after failures.[27, 28] They focus on durability and placement rather than per-operation authorization. Distributed data planes like Longhorn and Mayastor add replica sets and automatic rebuild logic.[29, 30] The system described here can run alongside these tools by treating them as black-box block devices while continuing to enforce mutation through the filesystem gate above them. When DRBD is used in this design, it is a per-replica hot-standby volume mirror; replica ordering and sealing remain a higher-level chain protocol.

Finally, TPM-based sealed storage and confidential VMs protect secrets and memory from some parts of the stack. A TPM can seal keys to boot measurements so that only an expected host configuration can unwrap them.[31] Confidential VMs protect guest memory from the hypervisor.[32, 33] In this design, these mechanisms can be used to harden the authorizer and to protect node identities but not to hold tenant data keys on storage nodes. Keeping decryption keys off the storage nodes and placing mutation control in the kernel is the core of the zero trust approach to the storage layer.

Taken together, the system can be seen as a storage-specific instantiation of zero trust: every sensitive mutation, such as edits and deletes of sealed content and node level administrative actions in the default profile or all write-like mutations in the strict profile, is an explicit, one use decision by a small TEE component; the kernel enforces that decision across all write paths; data is immutable by default after commit; node reboots are fenced by a per-node boot counter

with hardware-backed anti-rollback; and decryption keys never reside on the servers that store the ciphertext. Client side encryption remains the main defence against data leakage while the kernel gate and sealing reduce the damage from host compromise.

References

[1] Linux Security Module (LSM) Framework. https://www.kernel.org/doc/html/latest/security/lsm.html.

[2] SELinux Project Documentation. https://selinuxproject.org/page/Main_Page.

[3] fs-verity: File-based authenticity protection. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html.

[4] XFS Users Guide and Admin Notes. https://docs.kernel.org/filesystems/xfs.html.

[5] Verizon Data Breach Investigations Report 2024. https://www.verizon.com/business/resources/reports/dbir.

[6] CVE-2023-4966 Citrix Bleed Advisories. https://www.cisa.gov/news-events/alerts/2023/11/.

[7] Ivanti Connect Secure and Policy Secure 2024 Zero-day Advisories. https://www.cisa.gov.

[8] Okta Security Incident Support Case Management System 2023. https://sec.okta.com/articles/2023/10/.

[9] CISA Alert on Snowflake-related Customer Intrusions 2024. https://www.cisa.gov/news-events/alerts.

[10] SEC filings and reports on MGM and Caesars 2023 incidents. https://www.sec.gov/.

[11] CISA Advisory on ESXiArgs 2023. https://www.cisa.gov/news-events/alerts/2023/02/.

[12] Progress MOVEit Transfer CVE-2023-34362 and related. https://www.progress.com/security.

[13] Veeam Backup and Replication CVE-2023-27532. https://www.veeam.com/kb4424.

[14] Change Healthcare Incident Updates 2024. https://www.unitedhealthgroup.com/.

[15] QNAP and ASUSTOR Ransomware Advisories for Qlocker and DeadBolt. https://www.qnap.com/en/security-advisory.

[16] Microsoft Storm-0558 Token Forgery Analysis 2023. https://www.microsoft.com/security/blog/.

[17] UEFI Secure Boot Overview. https://uefi.org/specifications.

[18] Kernel Lockdown Documentation. https://www.kernel.org/doc/html/latest/admin-guide/lockdown.html.

[19] Linux IMA and EVM Integrity Subsystem. https://www.kernel.org/doc/html/latest/.

[20] Extended Attributes in Linux. https://www.kernel.org/doc/html/latest/filesystems/.

[21] fallocate(2) Linux man page. https://man7.org/linux/man-pages/man2/fallocate.2.html.

[22] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. OSDI 2004.

[23] OP-TEE Secure Storage Architecture. https://optee.readthedocs.io/en/latest/architecture/secure_storage.html.

[24] dm-verity Device-mapper target. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html.

[25] Amazon S3 Object Lock. https://docs.aws.amazon.com/AmazonS3/latest/dev/object-lock.html.

[26] Oracle or OpenZFS Administration Guide for Snapshots and Clones. https://openzfs.github.io/openzfs-docs/.

[27] DRBD Documentation. https://docs.linbit.com/docs/.

[28] LINSTOR Controller Documentation. https://docs.linbit.com/docs/linstor-guide/.

[29] Longhorn Architecture. https://longhorn.io/docs/.

[30] OpenEBS Mayastor Documentation. https://openebs.io/.

[31] TCG TPM 2.0 Library and Sealing. https://trustedcomputinggroup.org/.

[32] AMD SEV-SNP Security Overview. https://www.amd.com/en/technologies/secure-encrypted-virtualization.

[33] Intel TDX Overview. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/.

[34] Sahai and Waters. Attribute-Based Encryption foundational and survey resources. https://eprint.iacr.org/.

[35] H. Krawczyk and P. Eronen. RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function HKDF. https://www.rfc-editor.org/rfc/rfc5869.

[36] BLAKE3 One Function, Fast Everywhere. https://github.com/BLAKE3-team/BLAKE3.

[37] NIST SP 800-38D Recommendation for GCM. https://csrc.nist.gov/publications/detail/sp/800-38d/final.

[38] XChaCha20-Poly1305 draft and libsodium documentation. https://datatracker.irtf.org/doc/draft-irtf-cfrg-xchacha/.

[39] NIST SP 800-57 and SP 800-130 Key Management and PKI Considerations. https://csrc.nist.gov/publications.

[40] Key Rotation Operational Guidance. https://cloud.google.com/kms/docs/key-rotation.

[41] LINBIT DRBD 9 User's Guide. https://docs.linbit.com/docs/users-guide-9.0/.

[42] drbdsetup8 Manual. https://docs.linbit.com/man-pages/drbdsetup/.

[43] DRBD Quorum and on no quorum Behavior. https://docs.linbit.com/docs/users-guide-9.0/#s-quorum.

[44] DRBD Split-Brain Handling. https://docs.linbit.com/docs/users-guide-9.0/#s-split-brain.

[45] DRBD Online Verify. https://docs.linbit.com/docs/users-guide-9.0/#s-online-verify.

[46] DRBD Discard and Zeroing Semantics. https://docs.linbit.com/docs/users-guide-9.0/#s-discard-support.

[47] W3C Web Authentication WebAuthn. https://www.w3.org/TR/webauthn-2/.

[48] FIDO Alliance Client to Authenticator Protocol CTAP. https://fidoalliance.org/specifications/.

[49] OpenID Connect Core 1.0. https://openid.net/specs/openid-connect-core-1_0.html.

[50] SPIFFE Secure Production Identity Framework for Everyone. https://spiffe.io/.

[51] NIST FIPS 201 Personal Identity Verification PIV. https://csrc.nist.gov/publications/detail/fips/201/3/final.

[52] Microsoft Corporation. Microsoft SMB Protocol Versions 2 and 3 (MS-SMB2). https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/.

[53] T. Haynes and D. Noveck. RFC 7530: Network File System (NFS) Version 4 Protocol. https://www.rfc-editor.org/rfc/rfc7530.

[54] T. Haynes and D. Noveck et al. RFC 5661: Network File System (NFS) Version 4 Minor Version 1 Protocol. https://www.rfc-editor.org/rfc/rfc5661.

[55] M. Eisler. RFC 2203: RPCSEC_GSS Protocol Specification. https://www.rfc-editor.org/rfc/rfc2203.

[56] Linux Kernel Documentation. RPCSEC_GSS and Kerberos for NFS. https://www.kernel.org/doc/html/latest/filesystems/nfs/index.html.

[57] CITI NFSv4 RPCSEC_GSS Resources. http://www.citi.umich.edu/projects/nfsv4/linux/.