KONTOR STORAGE PROTOCOL

November 11, 2025

Contents

1 Introduction

The Kontor Storage Protocol is a system for ensuring that a set of untrusted actors are continuously and correctly storing data that they have publicly committed to preserving. This protocol implements a proof-of-retrievability (PoR) scheme on a blockchain coupled with crypto-economic incentives to provide scalable, perpetual storage that complements the state-machine replication architecture of the blockchain network within which the file storage system runs as a smart contract.

This protocol is a core feature of the Kontor Bitcoin metaprotocol. Kontor Indexers implement the greater Kontor protocol and execute the contracts that define the rules of the storage system. Each Indexer acts as a **Verifier**, independently verifying the integrity and validity of cryptographic proofs—published to the Bitcoin blockchain by a Storage Node (the **Prover**)—that claim to demonstrate that the node in question possesses a copy of certain data when it generates the proof in question.

Storage nodes are challenged pseudo-randomly by the Indexers using a shared source of entropy (in Kontor, the Bitcoin block hash). With each block, the Indexers use this shared entropy as a seed for the pseudo-random selection of file-node pairs, and the chosen Storage Nodes must respond to each of these **Challenges** by publishing a proof that they possess a copy of a pseudo-random subset of the file data that they have previously agreed to store. If a Storage Node fails to produce a valid proof within the allotted timeframe, the node is subject to a slashing of their escrowed balance of the KOR cryptocurrency as recorded by each Indexer.

The cryptographic proof system, implemented in the `Kontor-Crypto` Rust library, uses the Nova[1] recursive SNARKs via the `arecibo`[2] library. The proofs are constant-size (~10 kB) when aggregated over any number of files. These proofs are efficient to verify (approximately 50ms), making it feasible for the Kontor system to provide strong guarantees for decentralized data storage at scale.

2 Protocol

2.1 Economic Context: KOR and the Storage System

The Kontor storage protocol operates within a broader economic framework where the KOR token serves dual purposes: incentivizing perpetual data storage and fueling smart contract execution. Unlike Bitcoin, KOR is fundamentally inflationary by design.

Emissions and Storage Incentives: New KOR tokens are continuously minted and distributed to storage nodes as rewards for maintaining file agreements. These emissions scale with the total KOR supply and are weighted by file characteristics (size and creation order), ensuring that storage nodes earn predictable returns for their service. The emission rate adjusts dynamically based on network health, increasing when data replication falls below target levels.

Burn Mechanisms: Multiple deflationary mechanisms offset KOR inflation:

- Storage fees paid by users are entirely burned upon file creation
- Gas fees for smart contract execution are burned
- Penalties from slashing and voluntary exits are burned
- This creates economic pressure that balances emissions over time

Economic Balance: As the network matures, emissions from storage stabilize due to the protocol's tapering mechanism (earlier files receive higher perpetual rewards). Meanwhile, smart contract activity—the primary use case for KOR—grows independently and provides an ever-increasing source of burns. This architectural decision ensures that KOR can maintain stable monetary properties while supporting both perpetual storage and a thriving smart contract ecosystem.

The storage protocol described below implements the mechanics of this economic model, defining how emissions are calculated, distributed, and balanced against costs to create sustainable incentives for data preservation.

2.2 Summary

In the Kontor data storage protocol, **users** upload their data to **storage nodes**, which commit to storing their data forever. A user pays a one-time fee per file, which is calculated based on the file's size and the network's current state. This entire fee is burned.

An initial set of storage nodes are party to a file agreement upon its creation. These nodes are not paid from the user's fee; instead, they (and any nodes that join later) are compensated through ongoing emissions of the Kontor native token, KOR. The protocol uses a pooled stake model: to participate, a node must maintain a single, total KOR stake balance that is sufficient to cover all of its file storage commitments.

With the mining of each Bitcoin block, every Kontor indexer deterministically derives from the block hash a **challenge** that pseudo-randomly identifies a set of previously uploaded files to be audited. For each challenged file, one of its storage nodes is selected to publish a **proof** to the Bitcoin blockchain that it is indeed storing the data. This proof must be submitted within a fixed window of blocks.

If a storage node fails to produce a valid proof in time, a portion of its staked KOR is slashed. A part of the slashed funds is burned, and the remainder is distributed to the other nodes storing that same file. Conversely, nodes in a file agreement are rewarded each block a share of that file's KOR emissions.

After an agreement is created, nodes may join or leave it based on their operational costs and expected profits, as long as the file's replication level remains above a minimum threshold. Nodes pay a fee to leave based on (1) the quantity of KOR they escrowed to join the agreement and (2) the number of nodes in the agreement. Storage nodes are thus strongly incentivized to store all files that they have committed to.

2.3 State-Machine Replication

A blockchain operates by state-machine replication in which a Byzantine fault–tolerant consensus protocol is used by untrusted entities to agree on a log of events which are then executed deterministically to arrive at a shared state. A metaprotocol extends the model with the addition of a second state machine.

Each Kontor Indexer is deterministic and operates on an identical stream of input data (the Bitcoin blockchain); thus all correct Indexers act as a single effective Indexer that implements the protocol itself.

2.4 Actors

- **Users** $\mathcal{U} = \{u_1, ..., u_n\}$: A user is any account that stores data on the network. Users submit transactions to create file agreements and pay storage fees. They may also retrieve from storage nodes data uploaded by them or by other users.
- **Storage Nodes** $\mathcal{N} = \{n_1, ..., n_m\}$: A storage node is any account that commits to store files. Storage nodes submit transactions to join/leave file agreements and submit storage proofs.

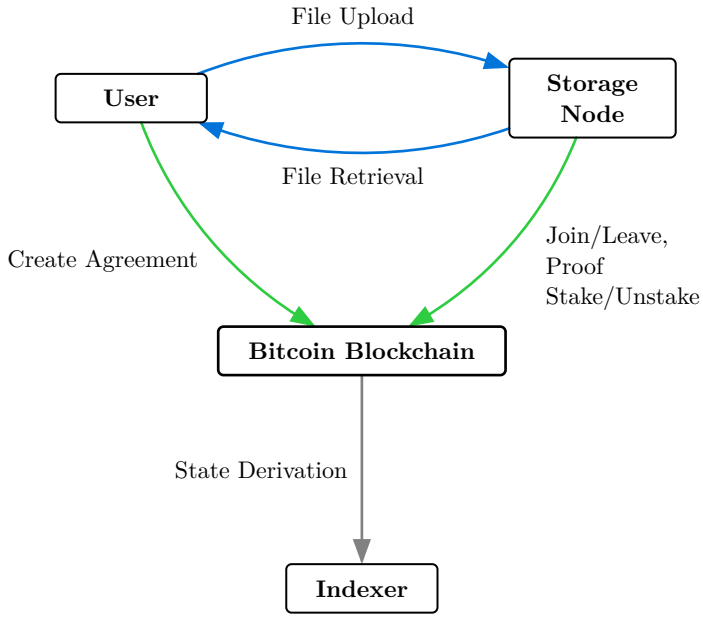

Figure 1: Actor Interaction Model
Blue: off-chain data transfer
Green: transactions
Grey: state derivation

2.5 Protocol Objects and State Variables

All state is maintained by Kontor indexers and updated deterministically as Bitcoin blocks are processed.

Time Semantics: Throughout this document, t denotes the current state of the protocol state machine. The value t increments with each state transition: transaction processing, block start, and block end. State variables like $\Omega(t)$ and $|\mathcal{F}(t)|$ represent current values at step t.

2.5.1 Global Protocol State

- $\mathcal{F}$: Set of active file agreements
- Pending agreements: File agreements awaiting activation
- $\mathcal{O}$: Set of active sponsorship offers
- $\mathcal{M}$: Set of active sponsorship agreements
- Sponsorship bond escrow: Mapping from (n_{entrant}, f) to escrowed bond amount (in KOR)
- Total files ever created (counter for assigning rank_f)
- For each file agreement $f \in \mathcal{F}$: $\mathcal{N}_f$ - the set of storage nodes storing file f
- For each node n: $\mathcal{F}_n$ - the set of active file agreements stored by node n
- Active challenges awaiting proofs

Activation Permanence: Once a file agreement enters $\mathcal{F}$ (by reaching $n_{\min}$ nodes), it NEVER leaves $\mathcal{F}$. The agreement remains active forever, even if all nodes are removed through slashing or stake insufficiency. In such cases, the agreement becomes under-replicated ($|\mathcal{N}_f| < n_{\min}$ or even $|\mathcal{N}_f| = 0$) but stays in $\mathcal{F}$ and continues to accrue emissions (though emissions are not minted when $|\mathcal{N}_f| = 0$, see Section 2.9.2). This ensures Ω is monotonically increasing and k_f calculations for new files remain deterministic.

All algorithm references to "files" or $\mathcal{F}$ mean ACTIVE files only. Pending agreements are tracked separately until activation.

2.5.2 Global Emission State

Indexers maintain a single global emission weight value:

- Ω - Total network emission weight, defined as:

$$\Omega \stackrel{\text{def}}{=} \begin{cases} 1.0 & \text{if } \mathcal{F} = \emptyset \\ \sum_{f \in \mathcal{F}} \omega_f & \text{otherwise} \end{cases} \tag{1}$$

  Initialized at genesis as $\Omega = 1.0$ to prevent division-by-zero. This is a single mutable global variable.

State Update Rule: When a file agreement f activates (reaches $n_{\min}$ nodes in `JoinAgreement`):

$$\Omega \leftarrow \Omega + \omega_f \tag{2}$$

Files never deactivate, so Ω is monotonically increasing. This value determines each file's share of network emissions and is used to calculate per-node base stakes for new files. When creating a new file agreement, use the current value of Ω (which reflects the state before this new file is added).

Edge Cases: The genesis value $\Omega = 1.0$ enables deterministic stake calculations when the first file is created. If all nodes leave a file agreement ($|\mathcal{N}_f| = 0$), the file remains in $\mathcal{F}$ with its ω_f counted in Ω, but emissions for that file are not minted (see Section 2.9.2). This maintains determinism in fee and stake calculations for new files. See [3, § Economic Analysis] for economic derivation.

2.5.3 Account Balances

Each account (user or storage node) has an address id and KOR balance in one of two states:

- **Spendable KOR** (b): Can be transferred, used to pay fees (storage fees, leave fees), or deposited into stake.
- **Staked KOR** (k_n, storage nodes only): Locked as security deposit to cover file agreements. Cannot be transferred and is subject to slashing. The protocol uses a pooled stake model where each node maintains a single stake balance k_n that covers all their file agreements. There is no per-agreement stake. [3]

2.5.4 Files

A file F consists of raw data that has been prepared for storage using erasure coding and Merkle tree commitment:

$$\begin{aligned} F = \big(&\mathrm{id}_f, \text{unique identifier} \\ &\text{data}, \text{raw file bytes} \\ &\text{size}, |\text{data}| \text{ in bytes} \\ &\rho, \text{Merkle root commitment} \\ &n_{\text{symbols}}, \text{number of data symbols (31-byte units)} \\ &n_{\text{codewords}}, \text{number of RS codewords} \\ &n_{\text{total}}, \text{total symbols including parity} \\ &\mathcal{T} \text{ Merkle tree over all symbols}\big) \end{aligned} \tag{3}$$

The file preparation function transforms raw data into a prepared file with metadata:

$$\text{Prepare-File} : \{0,1\}^* \to \mathcal{F}_{\text{prep}} \times \mathcal{M} \tag{4}$$

where $\mathcal{F}_{\text{prep}}$ contains the Merkle tree and $\mathcal{M}$ is metadata stored on-chain.

2.5.5 File Agreements

A file agreement $\mathcal{A}$ represents the protocol's commitment to store a specific file. An agreement is created when a user pays the storage fee and becomes active when $n_{\min}$ storage nodes join:

$$\begin{aligned}\mathcal{A} = \big(&\mathrm{id}_a, \text{agreement identifier}\\ &\text{file\_id}, \text{file being stored}\\ &M, \text{file metadata (root, size, etc.)}\\ &\mathcal{N}_a \subseteq \mathcal{N}, \text{set of storing nodes}\\ &\text{creation\_block}, \text{block height when created}\\ &\mathrm{rank}_f \in \mathbb{N}^+, \text{file creation order (immutable)}\\ &\omega_f \in \mathbb{R}^+, \text{file emission weight (immutable)}\\ &k_f \in \mathbb{R}^+, \text{per-node base stake for this file (in KOR, immutable)}\\ &\text{active true when } |\mathcal{N}_a| \geq n_{\min}\big)\end{aligned} \tag{5}$$

The values stored in the agreement structure (rank_f, ω_f, k_f) are computed at creation time:

1. **File rank**: rank_f = total number of files ever created + 1 (sequential counter)
2. **File emission weight**: $\omega_f = \frac{\ln\left(s_f^{\text{bytes}}\right)}{\ln(1+\mathrm{rank}_f)}$
3. **Network emission weight**: Retrieve current Ω from global state
4. **Per-node base stake**: $k_f = \left(\frac{\omega_f}{\Omega}\right) \cdot c_{\text{stake}} \cdot \ln\left(1 + |\mathcal{F}\frac{|}{F_{\text{scale}}}\right)$

Note: These calculations use the current network state (before this file is added to $\mathcal{F}$ or Ω).

See [3, § Protocol Functions] for economic derivation.

For each agreement, the protocol tracks:

- rank_f - immutable creation order (1 for first file, 2 for second, etc.)
- ω_f - file emission weight (determines share of network emissions)
- s_f^{bytes} - file size in bytes
- $n_{\text{symbols},f} = \left\lceil \frac{s_f^{\text{bytes}}}{31} \right\rceil$ - number of data symbols
- $n_{\text{codewords},f} = \left\lceil \frac{n_{\text{symbols},f}}{231} \right\rceil$ - number of RS codewords
- $n_{\text{total},f} = n_{\text{codewords},f} \times 255$ - total symbols including parity
- k_f - per-node base stake for this file (in KOR)
- $\mathcal{N}_f$ - set of storage nodes storing this file agreement

2.5.6 Sponsorship Offers

A sponsorship offer $o \in \mathcal{O}$ is a public, on-chain commitment by an existing storage node to sponsor a specific entrant for a file agreement:

$$\begin{aligned}o = \big(&\mathrm{id}_o, \text{offer identifier}\\ &f, \text{file agreement}\\ &n_{\text{sponsor}}, \text{offering node}\\ &n_{\text{entrant}}, \text{target entrant node}\\ &\gamma_{\text{rate}} \in [0,1], \text{commission rate}\\ &\gamma_{\text{duration}}, \text{duration in blocks}\\ &\beta_{\text{bond}}, \text{bond amount in KOR}\\ &\text{creation\_block}, \text{when offer created}\\ &\text{expiration\_block } \text{creation\_block} + W_{\text{offer}}\big)\end{aligned} \tag{6}$$

Offers are created via the `Create-Sponsorship-Offer` procedure and remain valid until either accepted by the entrant (converted to a sponsorship agreement) or expired. Expired offers are removed during block-end processing.

2.5.7 Sponsorship Agreements

A sponsorship agreement $m \in \mathcal{M}$ is an active arrangement where a node (entrant) receives emissions commission from a sponsor for a file agreement. Sponsorship agreements are created when an entrant accepts a sponsorship offer as part of the `Join-Agreement` procedure:

$$m = \left(f, n_{\text{entrant}}, n_{\text{sponsor}}, \gamma_{\text{rate}}, \gamma_{\text{duration}}, \beta_{\text{bond}}, t_{\text{start}}, \text{first\_proof\_complete}\right) \tag{7}$$

where:

- f - the file agreement being sponsored
- n_{entrant} - the node joining via sponsorship
- n_{sponsor} - the existing node providing the file data
- $\gamma_{\text{rate}} \in [0, 1]$ - fractional commission rate paid to sponsor
- γ_{duration} - duration in blocks
- β_{bond} - bond amount in KOR (held in escrow until first proof)
- t_{start} - activation block height
- first_proof_complete - boolean flag, set to true after entrant successfully proves first challenge for this file

The sponsorship is active from block t_{start} through $t_{\text{start}} + \gamma_{\text{duration}} - 1$, expiring when $t \geq t_{\text{start}} + \gamma_{\text{duration}}$.

Creation Mechanism: Sponsorship agreements are created through a trustless bond-escrow process:

1. Sponsor posts a public sponsorship offer on-chain via `Create-Sponsorship-Offer`, specifying commission terms and required bond amount
2. Entrant accepts via `Join-Agreement`, which atomically: (a) locks the bond in escrow, (b) creates the sponsorship agreement, (c) adds the entrant to the file agreement
3. Sponsor transfers file data off-chain after offer acceptance
4. Resolution occurs when the entrant is first challenged for this file:
    - If entrant proves successfully: bond is returned to entrant, sponsorship continues normally
    - If entrant fails first challenge: bond is transferred to sponsor (compensating fees and bandwidth costs), sponsorship voids retroactively (no commission ever paid), entrant is slashed normally

This bond mechanism makes the protocol fully trustless: the sponsor cannot extort (terms fixed on-chain first), the entrant cannot grief (bond at risk), and both parties have symmetric incentives to perform honestly.

Commission Scope: The commission rate applies exclusively to emissions from the sponsored file f. For each block during the sponsorship period:

- Entrant receives: $(1 - \gamma_{\text{rate}}) \times \varepsilon_f \frac{t}{|\mathcal{N}_f|}$ from file f
- Sponsor receives: $\left(\gamma_{\text{rate}} \times \varepsilon_f \frac{t}{|\mathcal{N}_f|}\right)$ additional from file f, plus full rewards from other files

The commission does not affect rewards from other files stored by the entrant, and slashing penalties are borne entirely by the slashed node without commission sharing.

2.5.8 Challenges

A challenge $\mathcal{C}$ is a deterministically generated event that requires a storage node to prove possession of specific file data within a window of blocks:

$$\begin{aligned}\mathcal{C} = \big(&\mathrm{id}_c, \text{unique challenge identifier}\\ &\text{node\_id}, \text{challenged storage node}\\ &\text{file\_id}, \text{file to prove possession of}\\ &M, \text{file metadata}\\ &\text{block\_height}, \text{creation block height}\\ &\text{expiration\_block}, \text{block\_height} + W_{\text{proof}}\\ &s, \text{number of symbols to prove}\\ &\sigma \text{ random seed for symbol selection}\big)\end{aligned} \tag{8}$$

The challenged symbols are sampled pseudo-randomly using the Bitcoin block hash as seed. If a file has fewer symbols than the protocol's sample size ($n_{\text{total},f} < s_{\text{chal}}$), all symbols are challenged.

Challenge Timing: A challenge created at block height h has expiration block $h + W_{\text{proof}}$. The challenged node must submit a valid proof transaction that is included in the blockchain by the END of block $h + W_{\text{proof}} - 1$. Expiration is checked in `Process-Failed-Challenges` during `On-Block-End`.

Challenge Frequency: Each file is selected for challenge with constant probability $p_f = \frac{C_{\text{target}}}{B}$ per block. For each selected file, one of its storing nodes is chosen uniformly at random. This ensures each file receives approximately C_{target} challenges per year regardless of network size.

For parameter values and economic analysis, see Parameter Selection and [3].

2.5.9 Storage Proofs

A storage proof π demonstrates that a node possesses file data at challenge time. A valid proof for challenge $\mathcal{C}$ is a Nova IVC proof demonstrating:

1. Possession of s randomly selected symbols from the committed file agreement data
2. Correct Merkle path verification for each challenged symbol
3. Consistency with the public Merkle root ρ

Nodes can aggregate multiple challenges into a single proof transaction to minimize Bitcoin fees. For the cryptographic construction, see Section 3.2. For economic analysis of proving costs and aggregation benefits, see [3].

2.5.10 Storage Node Operations

Indexer Requirements: Storage nodes must run Kontor indexers to participate in the protocol. The indexer maintains the complete protocol state by processing Bitcoin blocks deterministically, enabling nodes to:

- Track which file agreements they have joined
- Monitor incoming challenges directed at their node ID
- Maintain the current file ledger state (for proof generation)
- Determine optimal proof batching strategies
- Submit proof transactions at appropriate times

Without an indexer, a storage node cannot know when it has been challenged or what the current protocol state is. The indexer provides the authoritative view of all active challenges, file metadata, and expiration deadlines.

Proof Batching Autonomy: Storage nodes have complete autonomy in deciding which challenges to batch together and when to submit proof transactions. Within the expiration window (W_{proof} blocks), nodes can:

- Batch any subset of their pending challenges into a single proof
- Time their submissions strategically to optimize transaction fees
- Aggregate challenges from multiple files and multiple block heights
- Choose to respond to high-value challenges immediately while batching others

This autonomy enables nodes to optimize their operational costs. A node might batch many challenges into a single 10 kB proof transaction, amortizing the Bitcoin transaction fee across all challenges. The protocol imposes no requirements on batching strategy beyond the expiration deadline.

Multi-Batch Aggregation: The cryptographic proof system supports aggregating challenges with different parameters:

- **Different seeds**: Each challenge has its own seed σ derived from the block hash at challenge creation. Proofs can aggregate challenges with distinct seeds, enabling cross-batch aggregation.
- **Different block heights**: Challenges created at blocks $h_1, h_2, ..., h_k$ can be proven together, even if they span multiple blocks.
- **Different files**: Multi-file proofs naturally aggregate challenges across the node's entire storage portfolio.

2.5.11 Transactions and Procedures

Transactions are submitted to Bitcoin and processed deterministically by all indexers. Each transaction invokes one or more procedures with the authority of a signer. The storage protocol defines the following procedures:

- `Create-Storage-Agreement` - User creates file agreement
- `Join-Agreement` - Storage node joins file agreement (optionally accepting sponsorship offer)
- `Leave-Agreement` - Storage node voluntarily exits file agreement
- `Create-Sponsorship-Offer` - Storage node posts public offer to sponsor an entrant
- `Verify-Storage-Proof` - Storage node submits proof for challenge verification
- `Stake-Tokens` - Move spendable KOR to staked balance
- `Unstake-Tokens` - Move staked KOR to spendable balance

All procedures follow the signature: `Procedure(state, signer, ..., block_height)` where the middle parameters are procedure-specific.

2.6 Protocol Flow

The protocol operates in a cycle for each Bitcoin block:

1. **Block Start:** Generate challenges deterministically from block hash
2. **Transaction Processing:** Process user and storage node transactions
3. **Block End:** Process failed challenges, handle stake insufficiency, distribute emissions

2.6.1 File Agreement Creation Flow

The following sequence diagram shows the complete flow for creating a file agreement, from file preparation through activation.

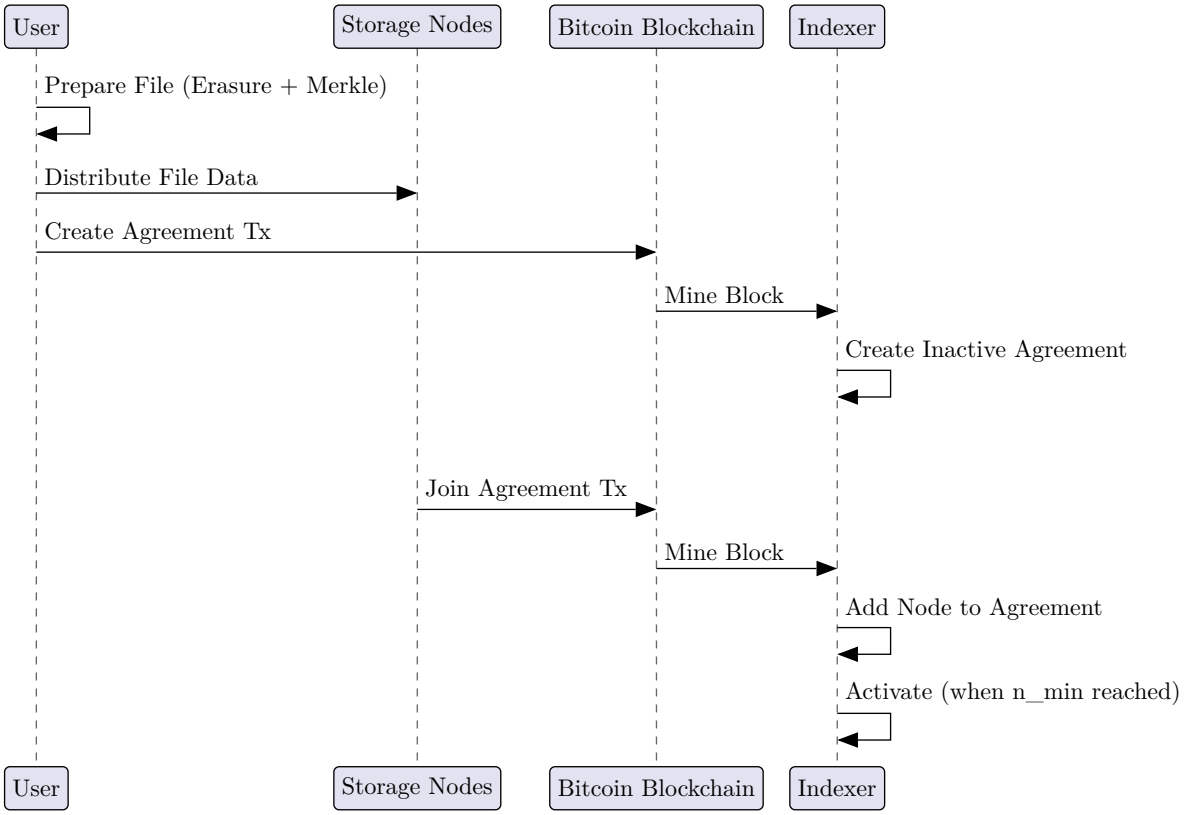

Figure 2: File Agreement Creation Flow. Users prepare files locally with erasure coding and Merkle tree commitment, distribute data to storage nodes off-chain, and broadcast the Create Agreement transaction. The agreement remains inactive until $n_{\min}$ storage nodes join, after which it activates and begins receiving emissions.

2.6.2 Challenge-Response Flow

The following sequence diagram shows the continuous challenge-response cycle that ensures storage nodes maintain possession of committed data.

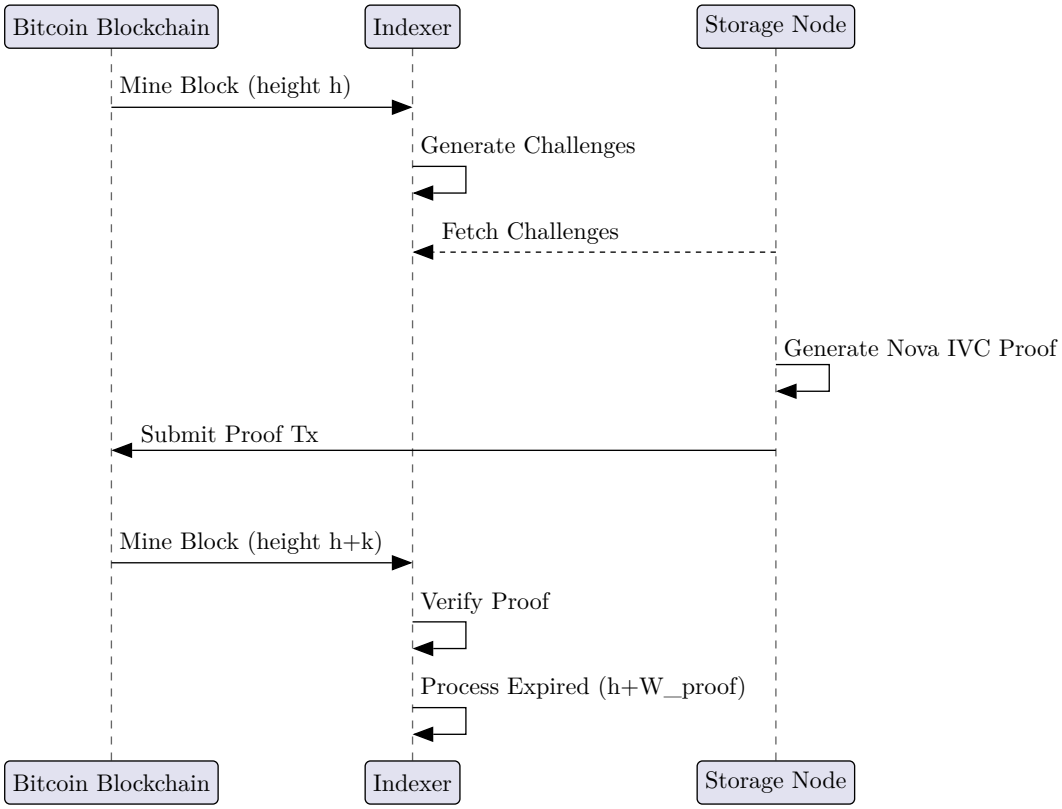

Figure 3: Challenge-Response Flow. Each block, indexers deterministically generate challenges from the block hash. Storage nodes fetch their challenges, generate Nova IVC proofs, and submit them within W_{proof} blocks or face slashing. At block end, the indexer processes expired challenges, slashes failed nodes, distributes penalties, and mints emissions to honest storers.

2.6.3 File Retrieval Flow

The following sequence diagram shows how users retrieve files from storage nodes through off-chain payment channels.

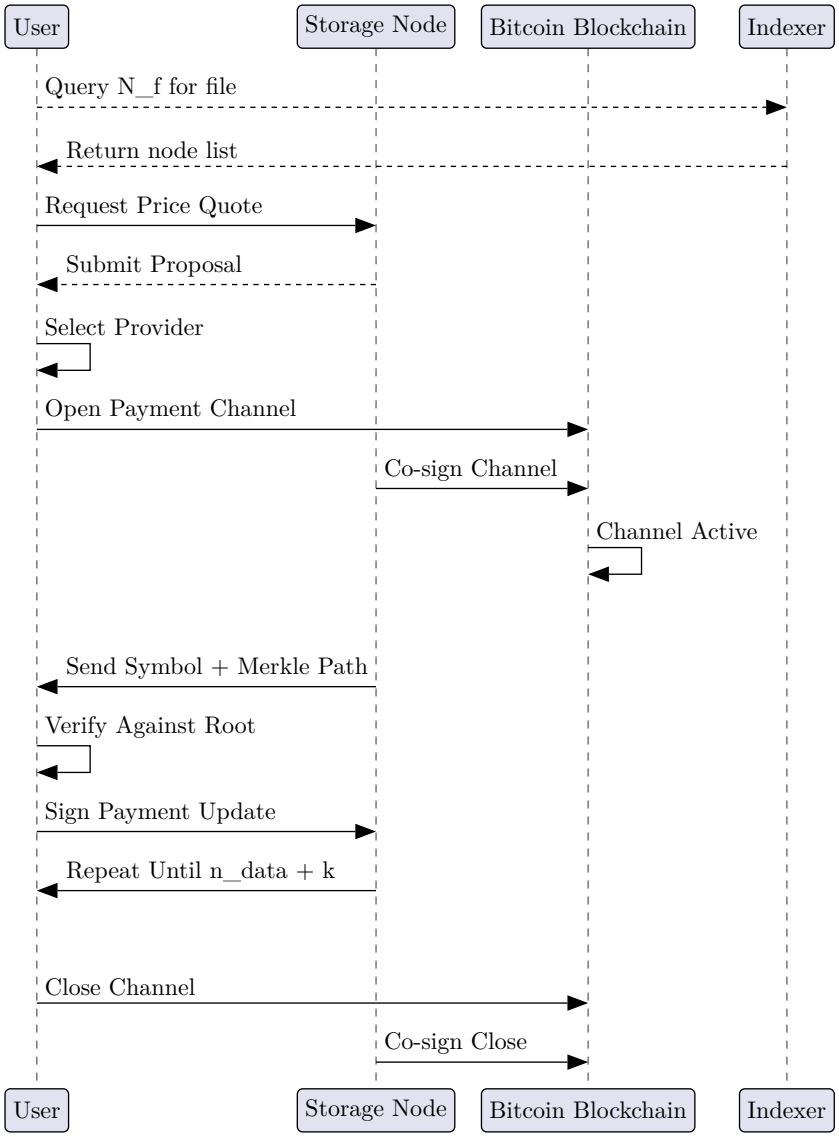

Figure 4: File Retrieval Flow. Users query indexer state for storage nodes ($\mathcal{N}_f$), negotiate pricing off-chain, establish Bitcoin payment channels, and perform atomic symbol-for-payment exchanges. Each symbol is verified against the on-chain Merkle root before payment. The erasure coding structure (each codeword requires 231 symbols minimum) solves the final-symbol problem: users request extra symbols beyond the reconstruction threshold, ensuring file recovery even if the provider withholds final symbols.

2.6.4 Block Start Processing

Algorithm 1: Block Start Processing

procedure ON-BLOCK-START(state, block_height, block_hash)
    ▷ Called at the start of each block before processing transactions
    ▷ Block has been mined; its hash is known
    ▷ Step 1: Generate challenges for this block
    $\mathcal{C}_{\text{new}} \leftarrow$ Generate-Challenges-For-Block(state, block_height, block_hash)
    ▷ Deterministically select files and nodes to challenge

    ▷ Step 2: Log challenge events

```
    if |C_new| > 0 then
        for C_id ∈ C_new do
            C ← state.challenges.get(C_id)
            if C ≠ ⊥ then
                ▷ Publish challenge created event for monitoring
                STATE.active_challenges.add(C)
            end
        end
    end
    return success
end
```

2.6.5 Transaction Processing

Between block start and block end, the protocol processes a sequence of transactions previously submitted to the mempool by users and storage nodes (indexers do not produce transactions) and included within the block by Bitcoin miners. Each transaction contains one or more procedure calls executed with the authority of a specific signer.

Algorithm 2: Transaction Processing

```
procedure PROCESS-TRANSACTIONS(state, block_height, transactions)
    ▷ Called after On-Block-Start, before On-Block-End
    ▷ Executes all transactions in block order
    for T ∈ transactions do
        ▷ Each transaction contains one or more procedure calls
        for P ∈ T.calls do
            ▷ Extract call parameters
            signer ← P.signer
            procedure ← P.procedure
            params ← P.params

            ▷ Dispatch to procedure with signature-specific parameters
            result ← procedure(state, signer, params, block_height)

            ▷ Validate result
            if result = ⊥ then
                ▷ Procedure call failed, transaction aborted
                return ⊥ (transaction failed)
            end
        end
    end
    return success
end
```

Each procedure call is validated for permissions (signer must match the actor), state validity, and economic constraints during execution.

2.6.6 Block End Processing

Algorithm 3: Block End Processing

```
procedure ON-BLOCK-END(state, block_height)
    ▷ Called at the end of each block after processing transactions
    ▷ Executes state transitions and economic updates
    ▷ Step 1: Process failed challenges (expired and invalid proofs)
    Process-Failed-Challenges(state, block_height)
    ▷ Identify expired challenges, slash nodes, distribute penalties

    ▷ Step 2: Handle stake insufficiency for all nodes
    for n ∈ state.all_nodes do
        k_n ← state.get_stake(n)
```

```
        F_n ← state.get_files_for_node(n)
        stake_sum ← Σ_{f∈F_n} state.get_agreement(f).base_stake
        λ_stake ← 1 + λ_slash / ln(2+|F_n|)
        k_required ← stake_sum · λ_stake
        ▷ Total required stake from economic model [3]
        if k_n < k_required then
            Handle-Stake-Insufficiency(n, state)
            ▷ Graceful exit or total forfeiture
        end
    end

    ▷ Step 3: Distribute emissions
    total_emitted ← Distribute-Storage-Rewards(state, block_height)
    ▷ Mint and distribute KOR to storing nodes

    ▷ Step 4: Expire sponsorship offers that were not accepted
    for o ∈ state.offers do
        if block_height ≥ o.expiration_block then
            state.offers.remove(o)
            ▷ Offer expired without acceptance
        end
    end

    ▷ Step 5: Expire sponsorships that reached their duration
    for m ∈ state.sponsorships do
        if block_height ≥ m.t_start + m.γ_duration then
            state.sponsorships.remove(m)
            ▷ Sponsorship commission period ended
        end
    end
    return success
end
```

2.7 Off-Chain Flows

These flows represent voluntary actions taken by users and storage nodes. Transaction creation is voluntary and economically motivated. See [3] for the economic incentives that drive these behaviors.

2.7.1 File Upload

Before creating a file agreement on-chain, users perform off-chain preparation and distribution of file data to potential storage nodes. This preparation phase transforms raw data into a fault-tolerant, self-authenticating structure that storage nodes can independently verify and use to generate proofs when challenged.

The upload process has three stages: (1) encode the file with erasure coding and build a Merkle commitment, (2) distribute the original file and metadata to storage nodes off-chain, and (3) broadcast a contract procedure call to create the on-chain agreement. Storage nodes independently prepare the file using the protocol's deterministic algorithm, ensuring all nodes storing a file compute identical Merkle trees without trusting the user's computation.

2.7.1.1 File Preparation

The file preparation algorithm transforms raw data into a structure optimized for challenge-based proof-of-retrievability.

Merkle Tree Commitment:

The protocol requires a cryptographic commitment enabling verification of individual data units without requiring the entire file. Merkle trees provide: constant-size commitment, logarithmic proof size, no trusted setup, and efficient verification in SNARK circuits.

- **Succinct commitment**: A single root hash (32 bytes) commits to arbitrary amounts of data
- **Selective opening**: Proving possession of one unit requires only a Merkle path (d sibling hashes), where $d = O(\log n)$
- **Binding**: Cryptographic collision-resistance ensures nodes cannot equivocate about file contents

Design decisions:

- **Hash function**: Poseidon is optimized for arithmetic circuits, requiring hundreds of constraints per invocation versus thousands for SHA-256 or Blake2, reducing IVC proving cost by orders of magnitude.
- **Symbol-leaf correspondence**: Each 31-byte symbol (data or parity) encodes to exactly one Merkle leaf, ensuring 1:1 alignment between symbols and tree leaves.
- **Binary structure**: Two children per node. Higher-arity trees (quaternary, octal) reduce depth but increase proof size (more siblings per level) and circuit complexity (multiple hash inputs per verification step).
- **Odd-node handling**: When a tree level has an odd number of nodes, the final node is duplicated (hashed with itself) to maintain uniform circuit structure. This ensures all internal nodes result from hashing two children.
- **Domain separation**: Distinct tags separate leaf encoding from internal node hashing, preventing cross-layer collision attacks.

Symbols and Field Element Encoding:

Files are partitioned into fixed 31-byte symbols. This size is the maximum that fits within a Pallas scalar field element (255 bits), enabling symbols to encode directly as Merkle leaves via little-endian byte representation with no intermediate hashing.

This direct encoding is critical for proof-of-retrievability. Challenges specify random symbol indices derived from the block hash. To prove symbol at index i, a node must possess the actual 31 bytes at that position, encode them as a field element, and provide the Merkle path. The field element encoding is reversible - the original 31 bytes can be extracted via inverse decoding. An attacker storing only the Merkle tree (field elements) without underlying bytes cannot answer challenges, because the SNARK circuit verification requires demonstrating knowledge of the bytes that encode to each challenged leaf.

Reed-Solomon Encoding:

Symbols are encoded using Reed-Solomon over GF(2^8) in a multi-codeword structure. Each codeword contains 231 data symbols and generates 24 parity symbols (10% overhead), totaling 255 symbols. The GF(2^8) field imposes a 255-symbol maximum per codeword. Files larger than 231 symbols (7,161 bytes) use multiple independent codewords. The GF(2^8) field is chosen for encoding speed; larger fields would reduce codeword count but slow encoding, an acceptable trade-off for one-time file preparation.

Erasure coding ensures that files passing challenges remain fully retrievable. Random sampling provides probabilistic detection: a node missing fraction ν of symbols has detection probability $1 - (1 - \nu)^s$ per challenge. This creates a gap where small losses may go undetected yet render files unrecoverable. Reed-Solomon encoding closes this gap - each codeword tolerates loss of up to 10% of its symbols while remaining reconstructible. Nodes can answer challenges for missing symbols by Reed-Solomon decoding from other symbols in the same codeword. The 10% parity overhead provides a safety margin: files remain retrievable when nodes retain sufficient symbols to avoid detection.

Reed-Solomon codes provide three properties: Maximum Distance Separable (any 231 of 255 symbols suffice for reconstruction), per-codeword independence (graceful degradation for

multi-codeword files), and systematic encoding (data symbols preserved unchanged, parity appended).

Encoding Procedure:

For file of size s_f^{bytes}:

1. **Partition**: $n_{\text{symbols}} = \left\lceil \frac{s_f^{\text{bytes}}}{31} \right\rceil$ data symbols (final symbol zero-padded to 31 bytes)
2. **Group**: $n_{\text{codewords}} = \left\lceil \frac{n_{\text{symbols}}}{231} \right\rceil$ codewords (≤231 data symbols each)
3. **Reed-Solomon encode** (GF(2^8)): Each codeword generates 24 parity symbols → $n_{\text{total}} = n_{\text{codewords}} \times 255$ total symbols
4. **Merkle tree**: Pad to $n' = 2^{\lceil \log_2 n_{\text{total}} \rceil}$, encode symbols as field elements, build Poseidon binary tree with depth $d = \log_2 n'$

Example: 1 MB file → 33,826 data symbols → 147 codewords → 37,485 total symbols → 65,536 padded → depth 16.

Reconstruction:

Given available symbols (some possibly missing), the decoder groups symbols into codewords (every 255 consecutive symbols). For each codeword with ≥231 available symbols, Reed-Solomon decoding recovers missing symbols and extracts the 231 data symbols. Codewords with fewer than 231 symbols cannot be reconstructed. The file is reassembled by concatenating data symbols from successfully reconstructed codewords and truncating to original size. This enables partial file recovery when data loss is concentrated in specific codewords.

Outputs:

File identifier: $\text{id}_f = \mathcal{H}_{\text{SHA256}}(\text{data})$

Off-chain (storage nodes): All n_{total} symbols (31 bytes each) + Merkle tree $\mathcal{T}$

On-chain (blockchain): Merkle root ρ, file identifier, padded leaf count n', original size s_f^{bytes}, filename

Constraints: $s_{\min} = 10 \text{ KB} \leq s_f^{\text{bytes}} \leq s_{\max} = 100 \text{ MB}$

Algorithm 4: File Preparation

procedure PREPARE-FILE(data, filename)
  ▷ Step 1: Compute file identifier
  $\text{id}_f \leftarrow \mathcal{H}_{\text{SHA256}}(\text{data})$
  $s_f^{\text{bytes}} \leftarrow |\text{data}|$

  ▷ Step 2: Partition into 31-byte symbols
  $n_{\text{symbols}} \leftarrow \left\lceil \frac{s_f^{\text{bytes}}}{31} \right\rceil$
  $\boldsymbol{D} \leftarrow \mathcal{P}(\text{data}, n_{\text{symbols}}, 31)$
  ▷ Partition into symbols, zero-pad final symbol to 31 bytes

  ▷ Step 3: Apply multi-codeword Reed-Solomon encoding
  $n_{\text{codewords}} \leftarrow \left\lceil \frac{n_{\text{symbols}}}{231} \right\rceil$
  ▷ Group symbols into codewords of 231 data symbols each
  $\boldsymbol{S} \leftarrow$ empty list
  ▷ Will hold all symbols: data + parity from all codewords
  for cw $\in [1, n_{\text{codewords}}]$ **do**
    $\boldsymbol{D}_{\text{cw}} \leftarrow$ symbols for this codeword (231 or remainder)
    $\boldsymbol{P}_{\text{cw}} \leftarrow \text{RS-Encode}_{\text{GF}(2^8)}(\boldsymbol{D}_{\text{cw}})$
    ▷ Generate 24 parity symbols for this codeword
    $\boldsymbol{S}$.extend($\boldsymbol{D}_{\text{cw}}$)
    $\boldsymbol{S}$.extend($\boldsymbol{P}_{\text{cw}}$)
  end

$n_{\text{total}} \leftarrow n_{\text{codewords}} \times 255$
▷ Each codeword contributes 255 symbols (231 data + 24 parity)

▷ Step 4: Pad to power-of-two count
$n' \leftarrow 2^{\lceil \log_2 n_{\text{total}} \rceil}$
while $|\boldsymbol{S}| < n'$ **do**
  $\boldsymbol{S}$.append($\mathbf{0}^{31}$)
end

▷ Step 5: Encode each symbol as Merkle leaf
$\boldsymbol{L} \leftarrow$ empty list
for $\boldsymbol{s} \in \boldsymbol{S}$ **do**
  $\ell \leftarrow \text{encode}_{\text{LE}(\boldsymbol{s})}$
  ▷ Direct little-endian encoding to field element
  $\boldsymbol{L}$.append(ℓ)
end

▷ Step 6: Build Merkle tree
$(\mathcal{T}, \rho) \leftarrow \mathcal{M}_{\text{Poseidon}}(\boldsymbol{L})$

▷ Step 7: Return prepared file and metadata
$F_{\text{prep}} \leftarrow \{\text{tree} : \mathcal{T}, \text{file\_id} : \text{id}_f, \text{root} : \rho\}$
$M \leftarrow \{\text{root} : \rho, \text{file\_id} : \text{id}_f, \text{padded\_len} : n', \text{original\_size} : s_f^{\text{bytes}}, \text{filename} : \text{filename}\}$
▷ Derived values: n_{chunks}, $n_{\text{codewords}}$, n_{total} computed from s_f^{bytes}
return (F_{prep}, M)
end

2.7.1.2 File Distribution

After preparing the file locally, the user distributes data to potential storage nodes off-chain. This distribution is independent of the on-chain agreement creation and may occur before, during, or after the agreement transaction is broadcast.

Data Transmitted: The user shares the original file bytes and the public metadata with each storage node. Nodes do not receive the user's prepared Merkle tree; instead, each node independently prepares the file using the protocol's deterministic preparation algorithm. This ensures nodes can verify data integrity without trusting the user's computation.

Verification by Recipients: When a storage node receives a file from a user, it performs the following verification:

1. Compute file identifier: $\text{id}'_f = \mathcal{H}_{\text{SHA256}}(\text{received data})$
2. Verify identifier matches metadata: $\text{id}'_f = \text{id}_f$
3. Run `Prepare-File` on received data with metadata's erasure configuration
4. Verify computed Merkle root matches metadata: $\rho' = \rho$

If verification succeeds, the node stores the prepared file structure (Merkle tree and all symbols). If verification fails, the node rejects the data. This independent preparation ensures all nodes storing a file compute identical Merkle trees and can generate consistent proofs.

User Incentives for Wide Distribution: Users benefit from sharing files with as many storage nodes as possible before creating the on-chain agreement:

- **Faster activation**: More nodes with the file data means the agreement can reach $n_{\min}$ nodes and activate more quickly, beginning to accrue anti-decay emissions sooner.
- **Reduced gatekeeping risk**: If only one or two nodes possess the file initially, they can extract monopoly rents through sponsorship agreements. Wide initial distribution creates a competitive market and prevents data monopolization.

- **Lower future sponsorship costs**: New nodes joining later can obtain the file from any existing storer. More initial nodes means more potential sponsors, driving sponsorship commission rates down through competition.

The user typically distributes files through direct peer-to-peer transfer, public portals, or other off-chain channels. The protocol does not enforce or verify off-chain distribution; it merely incentivizes it through economic mechanisms.

2.7.2 File Retrieval

File retrieval occurs entirely off-chain using Bitcoin payment channels with atomic symbol-for-payment exchanges. The protocol leverages the erasure coding structure to solve the sequential exchange problem where one party must accept risk on the final transfer.

Discovery and Negotiation: The user identifies storage nodes holding the target file by querying the protocol state for $\mathcal{N}_f$. The user contacts providers off-chain to request price quotes. Providers respond with pricing (typically in satoshis per symbol or per full file) and payment channel coordinates. The user selects one or more providers based on price, latency, and redundancy preferences.

Payment Channel Establishment: The user and selected provider(s) establish bidirectional Bitcoin payment channels using standard Lightning Network or similar technology. The user commits the agreed payment amount to the channel. This channel setup occurs entirely on the Bitcoin layer and does not involve the Kontor protocol state.

Atomic Symbol Exchange: The file preparation yields n_{symbols} data symbols. With erasure coding across $n_{\text{codewords}}$ codewords, each codeword requires 231 symbols minimum (90%) for reconstruction. The retrieval proceeds via atomic exchanges:

For each symbol $i \in \{1, ..., n_{\text{symbols}} + k\}$ where $k \geq 1$:

1. Provider sends symbol s_i (31 bytes) and its Merkle path π_i to user
2. User verifies Merkle path against on-chain root ρ: $\text{Verify-Merkle-Path}(\rho, s_i, \pi_i)$
3. If valid: User signs payment channel update transferring p_{symbol} to provider
4. If invalid: User aborts retrieval, closes channel with current state
5. Both parties sign the updated channel state

Final-Symbol Problem Resolution: In sequential exchanges without a trusted third party[4], one party must accept risk on the final transfer. The erasure coding structure eliminates this asymmetry: the user requests $n_{\text{symbols}} + k$ symbols where $k \geq 1$ represents redundancy beyond the reconstruction threshold. After receiving sufficient symbols per codeword (≥231 per codeword), the user can reconstruct the complete file. If the provider withholds the final k symbols:

- User loses: Payment for k symbols (minimal overhead)
- Provider loses: Payment for withheld symbols
- User obtains reconstructible file data regardless

The economic incentive is symmetric: withholding final symbols costs the provider more in lost payment than the user loses in redundancy overhead. In practice, providers deliver all symbols to maximize payment.

Channel Settlement: After complete file transfer and verification, both parties cooperatively close the payment channel, settling the final state on the Bitcoin blockchain. The channel can also remain open for future retrievals between the same parties, amortizing the channel open/close costs across multiple file transfers.

Trust Model: This retrieval mechanism is effectively trustless: cryptographic verification (Merkle paths) ensures data validity, and economic incentives (redundancy overhead) ensure completion. The protocol does not track or enforce retrieval agreements on-chain; file retrieval is a bilateral contract between user and provider settled through Bitcoin payment channels.

2.7.3 Sponsored Join Negotiation

When a node wishes to join a file agreement but cannot obtain the file data through public channels, it can request sponsorship from existing storers. The trustless two-step sponsorship mechanism ensures neither party can exploit the other.

Discovery Phase: The entrant identifies a target file agreement and queries the existing storers ($\mathcal{N}_f$) for sponsorship availability. This discovery happens off-chain through direct communication, public registries, or gossip protocols.

Offer Creation (On-Chain): A willing sponsor posts a public sponsorship offer by invoking `Create-Sponsorship-Offer`. The offer specifies:

- Target entrant (specific node identifier)
- Commission rate (γ_{rate}) and duration (γ_{duration})
- Required bond amount (β_{bond} in KOR)
- Expiration deadline (W_{offer} blocks from creation)

The sponsor pays Bitcoin transaction fees to broadcast this offer, creating a credible commitment with fixed terms visible to all indexers. The bond requirement protects the sponsor from griefing attacks.

Data Transfer (Off-Chain): After the offer is confirmed on-chain, the sponsor transfers the file data to the entrant off-chain. Transfer methods are implementation-specific (direct peer-to-peer, encrypted channels, etc.). The protocol does not observe or enforce this transfer.

Acceptance and Bond Escrow (On-Chain): If the entrant successfully receives and verifies the file data (runs `Prepare-File` and confirms root matches), it invokes `Join-Agreement` with the offer identifier. The procedure atomically:

- Validates the offer exists, targets this entrant, and has not expired
- Locks the bond (β_{bond}) in escrow from entrant's spendable balance
- Creates the sponsorship agreement with the offer's terms
- Adds the entrant to the file agreement
- Removes the accepted offer from state

The bond remains in escrow until the entrant's first challenge for this file is resolved.

Bond Resolution: When the entrant is first challenged for the sponsored file:

- **Success**: If the entrant submits a valid proof, the bond is returned to the entrant's balance and the sponsorship continues normally for its full duration.
- **Failure**: If the entrant fails the challenge (expires or invalid proof), the bond is transferred to the sponsor (compensating Bitcoin fees and bandwidth costs), the sponsorship is voided retroactively (no commission ever paid), and the entrant is slashed and removed normally.

This bond-escrow mechanism makes sponsorship fully trustless: the sponsor cannot extort (terms fixed on-chain first), the entrant cannot grief (bond at risk equals sponsor's costs), and both parties have symmetric incentives to perform honestly. If the sponsor posts an offer but never sends data, they lose Bitcoin fees while the entrant loses nothing (can reject or let offer expire).

Market Dynamics: Multiple sponsors may compete by posting offers with lower γ_{rate} for the same or different entrants. This competitive market prevents gatekeeping cartels: any existing member has incentive to defect and capture the commission. The equilibrium commission rate (derived in [3]) balances the sponsor's bandwidth cost against the NPV of commission payments:

$$\gamma_{\text{eq}}(D) \approx \frac{c_{\text{transfer}}^{\text{USD}} \cdot \rho \cdot \left(|\mathcal{N}_f| + 1\right)}{\varepsilon_f(t) \cdot \xi_{\text{KOR/USD}} \cdot \left(1 - (1+\rho)^{-D}\right)} \tag{9}$$

2.8 Transaction Processing

The protocol deterministically processes transactions that have been included in Bitcoin blocks. Each transaction contains one or more procedure calls that modify the protocol state. The following procedures can be invoked by users and storage nodes:

2.8.1 Create Storage Agreement

The user invokes the `Create-Storage-Agreement` procedure by broadcasting a contract call to the Bitcoin blockchain. This procedure call includes the file metadata (Merkle root, file identifier, erasure configuration, and size parameters) and is processed deterministically by all indexers when the containing Bitcoin transaction is included in a block.

File distribution (see Section 2.7.1.2) is an independent off-chain process that may occur before, during, or after agreement creation. Users typically distribute files to potential storage nodes before or concurrently with the on-chain agreement to enable faster activation, though the protocol does not enforce this ordering.

The procedure performs the following operations: (1) validates the metadata and file size constraints, (2) calculates the file's immutable economic parameters (rank, emission weight, per-node base stake) based on current network state, (3) collects the storage fee from the user and burns it, (4) creates the file agreement in an inactive state, and (5) initializes the membership tracking structures. The agreement remains inactive until $n_{\min}$ storage nodes join, at which point it activates and begins receiving emissions.

The storage fee υ_f is calculated deterministically from current network state: $\upsilon_f = \chi_{\text{fee}} \cdot k_f$ where $k_f = \left(\frac{\omega_f}{\Omega}\right) \cdot c_{\text{stake}} \cdot \ln\left(1 + |\mathcal{F}| \frac{}{F_{\text{scale}}}\right)$. The values of Ω and $|\mathcal{F}|$ are read from the protocol state at the block immediately before this agreement is created, ensuring deterministic and predictable fee calculation for all indexers.

Algorithm 5: Create Storage Agreement

procedure CREATE-STORAGE-AGREEMENT(state, signer, file_id, metadata, block_height)
  ▷ Invoked via contract procedure call in Bitcoin transaction
  ▷ User has distributed file data off-chain before broadcasting
  $M \leftarrow$ metadata
  ▷ Metadata includes: root hash, erasure config, sizes, filename

  ▷ Step 1: Compute agreement identifier
  $\text{id}_a \leftarrow \mathcal{H}(\text{file\_id} \parallel \text{block\_height})$

  ▷ Step 2: Calculate file rank and emission weight
  $\text{rank}_f \leftarrow \text{state.total\_files\_ever\_created} + 1$
  ▷ Sequential creation counter
  $s_f^{\text{bytes}} \leftarrow M.\text{size}$
  $\omega_f \leftarrow \frac{\ln(s_f^{\text{bytes}})}{\ln(1+\text{rank}_f)}$
  ▷ File emission weight from size and rank

  ▷ Step 3: Retrieve current network state
  $\Omega \leftarrow \text{state.get\_omega}()$
  $|\mathcal{F}| \leftarrow |\text{state.active\_files}|$
  ▷ Current global emission weight and file agreement count

  ▷ Step 4: Calculate per-node base stake
  $k_f \leftarrow \left(\frac{\omega_f}{\Omega}\right) \cdot c_{\text{stake}} \cdot \ln\left(1 + |\mathcal{F}| \frac{}{F_{\text{scale}}}\right)$
  ▷ Per-node base stake from economic model [3]
  $\upsilon_f \leftarrow \chi_{\text{fee}} \cdot k_f$
  ▷ Storage fee from economic model [3]

  ▷ Step 5: User pays storage fee (burned)
  if state.balance(signer) $< \upsilon_f$ **then**

return $\perp$ (insufficient balance)
end
state.burn_tokens(signer, υ_f)
▷ Storage fee is burned to create agreement

▷ Step 6: Increment file creation counter
state.increment_total_files_created()

▷ Step 7: Create agreement structure
$\mathcal{A} \leftarrow \{\text{id} : \text{id}_a, \text{user\_id} : \text{signer}, \text{file\_id} : \text{file\_id}, \text{metadata} : M, \text{nodes} : \emptyset, \text{creation\_block} : \text{block\_height}, \text{rank} : \text{rank}_f, \text{emission\_weight} : \omega_f, \text{base\_stake} : k_f, \text{active} : \text{false}\}$
▷ File agreement starts inactive until $n_{\min}$ nodes join

▷ Step 8: Store agreement
state.agreements.set($\text{id}_a, \mathcal{A}$)

▷ Step 9: Initialize membership sets for this file agreement
state.set_nodes_for_file(file_id, $\emptyset$)
return id_a
end

2.8.2 Create Sponsorship Offer

Existing storage nodes can post public sponsorship offers to facilitate new nodes joining file agreements. The sponsor invokes the `Create-Sponsorship-Offer` procedure via a contract call, committing to provide file data to a specific entrant in exchange for commission payments.

The procedure validates that the sponsor stores the target file, creates the offer with specified terms (commission rate and duration), and stores it in the active offer set with an expiration deadline. The sponsor incurs Bitcoin transaction fees to post the offer, creating a credible commitment before any data transfer occurs. This trustless design prevents sponsor extortion (terms are fixed on-chain) and entrant fraud (entrant only accepts after verifying data).

Algorithm 6: Create Sponsorship Offer

procedure CREATE-SPONSORSHIP-OFFER(state, signer, file_id, entrant_id, gamma_rate, gamma_duration, bond_am
▷ Sponsor posts public offer to sponsor specific entrant
▷ Pays Bitcoin tx fee as credible commitment
▷ Specifies bond amount to protect against griefing
▷ Step 1: Validate sponsor stores the file
$\mathcal{F}_n$ (sponsor's files) $\leftarrow$ state.get_files_for_node(signer)
if file_id $\neg \in \mathcal{F}_n$ **then**
return $\perp$ (sponsor does not store file)
end

▷ Step 2: Validate commission parameters
if $\gamma_{\text{rate}} \leq 0 \vee \gamma_{\text{rate}} > 1$ **then**
return $\perp$ (invalid commission rate)
end
if $\gamma_{\text{duration}} \leq 0$ **then**
return $\perp$ (invalid duration)
end
if bond_amount ≤ 0 **then**
return $\perp$ (invalid bond amount)
end
▷ Bond protects sponsor from griefing attacks

▷ Step 3: Create offer identifier
$\text{id}_o \leftarrow \mathcal{H}(\text{signer} \parallel \text{file\_id} \parallel \text{entrant\_id} \parallel \text{block\_height})$

```
    ▷ Step 4: Create offer structure
    o ← {id : id_o, file_id : file_id, sponsor : signer, entrant : entrant_id, rate : γ_rate, duration : γ_duration, bond : bond_amount, creation_block : block_height, expiration_block : block_height + W_offer}
    ▷ Offer expires after W_offer blocks if not accepted

    ▷ Step 5: Store offer in active offer set
    state.offers.add(o)
    return id_o
end
```

2.8.3 Join Agreement

Storage nodes join existing file agreements by invoking the `Join-Agreement` procedure via a contract call broadcast to the Bitcoin blockchain. Before broadcasting this call, nodes must obtain the file data and verify its integrity by independently running the file preparation algorithm and confirming the computed Merkle root matches the on-chain agreement metadata. While the protocol does not enforce this pre-verification (nodes can join without possessing valid data), without the data the node will fail challenges and be slashed, making pre-verification economically rational.

Nodes can join through two mechanisms:

- **Unsponsored join**: The node acquires file data through off-chain channels (from the user, public portals, or other sources) and joins directly. The node must have sufficient stake to cover the projected file set after joining.
- **Sponsored join**: The node accepts an existing sponsorship offer by providing the offer identifier. The procedure validates that the offer exists, is not expired, and targets this specific entrant, then converts the offer into an active sponsorship agreement while adding the node to the file agreement.

The procedure validates stake sufficiency, processes sponsorship offers if provided, adds the node to the file agreement, and activates the agreement when $n_{\min}$ nodes is reached.

Algorithm 7: Join Agreement

```
procedure JOIN-AGREEMENT(state, signer, agreement_id, offer_id, block_height)
    ▷ Entrant joins file agreement, optionally accepting sponsorship offer
    ▷ Should verify file data off-chain to avoid failing challenges
    ▷ Step 1: Retrieve file agreement
    A ← state.agreements.get(agreement_id)
    if A = ⊥ then
        return ⊥ (file agreement not found)
    end

    ▷ Step 2: Check if node already in agreement
    if signer ∈ A.nodes then
        return ⊥ (already joined)
    end

    ▷ Step 3: Process sponsorship offer if provided
    if offer_id ≠ ⊥ then
        ▷ Retrieve and validate offer
        o ← state.offers.get(offer_id)
        if o = ⊥ then
            return ⊥ (offer not found)
        end

        ▷ Validate offer is for this entrant and file
        if o.entrant ≠ signer ∨ o.file_id ≠ A.file_id then
            return ⊥ (offer mismatch)
```

```
    end

    ▷ Validate offer not expired
    if block_height ≥ o.expiration_block then
        return ⊥ (offer expired)
    end

    ▷ Validate sponsor still in agreement
    if o.sponsor ¬ ∈ A.nodes then
        return ⊥ (sponsor no longer in agreement)
    end
end

▷ Step 4: Verify stake requirement
F_n (node file agreements) ← state.get_files_for_node(signer)
F_n' (projected file agreements) ← F_n ∪ {A.file_id}
k_n (node stake) ← state.get_stake(signer)
▷ Calculate required stake with projected file set
stake_sum ← Σ_{f∈F_n'} state.get_agreement(f).base_stake
λ_stake ← 1 + λ_slash / ln(2+ |F_n'|)
k_required ← stake_sum · λ_stake
▷ Total required stake from economic model [3]
if k_n < k_required then
    return ⊥ (insufficient stake)
end
▷ Predictive check: stake validated using PROJECTED file set F_n'
▷ Node must have sufficient stake assuming join succeeds

▷ Step 5: Lock bond and create sponsorship if accepting offer
if offer_id ≠ ⊥ then
    ▷ Validate entrant has sufficient balance for bond
    b_n (entrant balance) ← state.get_balance(signer)
    if b_n < o.bond then
        return ⊥ (insufficient balance for bond)
    end

    ▷ Lock bond in escrow
    state.reduce_balance(signer, o.bond)
    state.set_bond_escrow((signer, A.file_id), o.bond)
    ▷ Bond held until first challenge resolution for this file

    ▷ Create sponsorship agreement from accepted offer
    m ← {file_id : A.file_id, entrant : signer, sponsor : o.sponsor, rate : o.rate, duration :
    o.duration, bond : o.bond, start : block_height, first_proof_complete : false}
    state.sponsorships.add(m)

    ▷ Remove accepted offer from active offers
    state.offers.remove(o)
end

▷ Step 6: Add node to file agreement
A.nodes.add(signer)

▷ Step 7: Update membership sets
N_f (nodes for file agreement) ← state.get_nodes_for_file(A.file_id)
N_f.add(signer)
state.set_nodes_for_file(A.file_id, N_f)
F_n (file agreements for node) ← state.get_files_for_node(signer)
F_n.add(A.file_id)
state.set_files_for_node(signer, F_n)
```

▷ Step 8: Activate file agreement if threshold reached
if $|\mathcal{A}.\text{nodes}| \geq n_{\min} \wedge \mathcal{A}.\text{active} = \text{false}$ **then**
  $\mathcal{A}.\text{active} \leftarrow \text{true}$
  $\mathcal{F}.\text{add}(\mathcal{A}.\text{file\_id})$
  ▷ File agreement added to $\mathcal{F}$, begins receiving emissions

  ▷ Update global emission weight
  $\omega_f \leftarrow \mathcal{A}.\text{emission\_weight}$
  $\Omega \leftarrow \text{state.get\_omega}()$
  $\Omega \leftarrow \Omega + \omega_f$
  $\text{state.set\_omega}(\Omega)$
  ▷ Increment global Ω when file agreement activates
end
return success
end

2.8.4 Leave Agreement

Storage nodes may voluntarily exit file agreements by invoking the `Leave-Agreement` procedure via a contract call. Voluntary departure is only permitted when the file agreement has more than the minimum required nodes ($|\mathcal{N}_f| > n_{\min}$) and the node has sufficient spendable balance to pay the leave fee.

The procedure validates the departure conditions, collects the leave fee $\varphi_{\text{leave}} = k_f \cdot \left(\frac{n_{\min}}{|\mathcal{N}_f|}\right)^2$ from the node's spendable balance and burns it, removes the node from the file agreement, updates membership tracking, and voids any sponsorship agreements where the departing node was either an entrant or a sponsor.

Algorithm 8: Leave Agreement

procedure LEAVE-AGREEMENT(state, signer, agreement_id, block_height)
  ▷ Step 1: Retrieve file agreement
  $\mathcal{A} \leftarrow \text{state.agreements.get}(\text{agreement\_id})$
  if $\mathcal{A} = \perp \vee \text{signer} \neg \in \mathcal{A}.\text{nodes}$ **then**
    return $\perp$ (invalid request)
  end

  ▷ Step 2: Check minimum nodes constraint
  $\mathcal{N}_f$ (nodes for file agreement) $\leftarrow \text{state.get\_nodes\_for\_file}(\mathcal{A}.\text{file\_id})$
  if $|\mathcal{N}_f| \leq n_{\min}$ **then**
    return $\perp$ (cannot leave: would violate minimum)
  end

  ▷ Step 3: Calculate and verify leave fee from spendable balance
  $k_f \leftarrow \mathcal{A}.\text{base\_stake}$
  $\varphi_{\text{leave}} \leftarrow k_f \cdot \left(\frac{n_{\min}}{|\mathcal{N}_f|}\right)^2$
  ▷ Leave fee from economic model [3]
  b_n (spendable balance) $\leftarrow \text{state.get\_balance}(\text{signer})$
  if $b_n < \varphi_{\text{leave}}$ **then**
    return $\perp$ (insufficient balance for leave fee)
  end

  ▷ Step 4: Burn leave fee from spendable balance
  $\text{state.reduce\_balance}(\text{signer}, \varphi_{\text{leave}})$
  $\text{state.burn}(\varphi_{\text{leave}})$

  ▷ Step 5: Remove node from file agreement
  $\mathcal{A}.\text{nodes.remove}(\text{signer})$

  ▷ Step 6: Update membership sets
  $\mathcal{N}_f.\text{remove}(\text{signer})$

```
    state.set_nodes_for_file(A.file_id, N_f)
    F_n (file agreements for node) ← state.get_files_for_node(signer)
    F_n.remove(A.file_id)
    state.set_files_for_node(signer, F_n)

    ▷ Step 7: Void sponsorships involving departing node
    for m ∈ state.sponsorships do
        if m.entrant = signer ∨ m.sponsor = signer then
            state.sponsorships.remove(m)
            ▷ Voided: node was sponsor or entrant
        end
    end
    return success
end
```

2.8.5 Verify Storage Proof

Storage nodes respond to challenges by invoking the `Verify-Storage-Proof` procedure via a contract call that includes the cryptographic proof and challenge identifier. The procedure extracts the challenge parameters and file metadata, reconstructs the expected public inputs, and verifies the Nova IVC proof using the Spartan verification algorithm.

If the proof is valid, the challenge is marked as verified and removed from the active challenge queue. If the proof is invalid or malformed, the challenge is added to the failed challenge queue, triggering slashing during block-end processing.

Algorithm 9: Verify Storage Proof

```
procedure VERIFY-STORAGE-PROOF(state, signer, challenge_id, proof, block_height)
    ▷ Step 1: Retrieve and validate challenge
    C ← state.challenges.get(challenge_id)
    if C = ⊥ then
        return ⊥ (challenge not found)
    end
    if signer ≠ C.node_id then
        return ⊥ (unauthorized - wrong node)
    end

    ▷ Step 2: Extract challenge parameters
    M (metadata) ← C.metadata
    ρ (expected root) ← M.root
    s (num sectors) ← C.num_sectors
    σ (seed) ← C.seed
    n′ (padded leaves) ← M.padded_len
    d (tree depth) ← trailing_zeros(n′)
    ▷ Depth derived from padded length (power of two)

    ▷ Step 3: Reconstruct public inputs
    s_0 (initial state) ← 0
    z_0 ← [ρ, s_0, σ, d]
    if proof.public_inputs ≠ z_0 then
        return ⊥ (public input mismatch)
    end

    ▷ Step 4: Verify Nova SNARK
    pp (public params) ← G(d, s)
    π_compressed ← proof.compressed_snark
    valid ← Spartan.Verify(pp, π_compressed, z_0, s)
    if ¬ valid then
        ▷ Invalid proof - add to failed queue for slashing
        STATE.failed_challenges.add(C.id)
```

```
    return ⊥ (proof verification failed)
  end

  ▷ Step 5: Validate final state consistency
  z_s ← proof.final_state
  ρ_final ← z_s[0]
  if ρ_final ≠ ρ then
    return ⊥ (final root mismatch)
  end

  ▷ Step 6: Mark challenge as verified and remove from active list
  STATE.verified_challenges.add(challenge_id)
  STATE.active_challenges.remove(C)

  ▷ Step 7: Release bond if this is first proof for sponsored join
  file_id ← C.file_id
  for m ∈ state.sponsorships do
    if m.file_id = file_id ∧ m.entrant = signer ∧ m.first_proof_complete = false then
      ▷ This is entrant's first successful proof for sponsored file

      ▷ Return bond to entrant
      bond ← state.get_bond_escrow((signer, file_id))
      state.add_balance(signer, bond)
      state.clear_bond_escrow((signer, file_id))
      ▷ Bond returned: entrant proved possession of valid data

      ▷ Mark first proof complete
      m.first_proof_complete ← true
      ▷ Sponsorship now unconditional
    end
  end
  return success
end
```

Algorithm 10: Stake Tokens

```
procedure STAKE-TOKENS(state, signer, amount, block_height)
  ▷ Validate amount
  b_n (spendable balance) ← state.get_balance(signer)
  if amount ≤ 0 ∨ amount > b_n then
    return ⊥ (invalid amount)
  end

  ▷ Move spendable KOR to staked balance
  state.reduce_balance(signer, amount)
  state.add_stake(signer, amount)
  return success
end
```

2.8.6 Stake Tokens

Storage nodes invoke the `Stake-Tokens` procedure to move KOR from their spendable balance to their staked balance. This increases the node's stake capacity, enabling it to join additional file agreements. The procedure validates the amount and balance, then transfers the specified amount from spendable to staked balance.

Algorithm 10: Stake Tokens

```
procedure STAKE-TOKENS(state, signer, amount, block_height)
  ▷ Validate amount
  b_n (spendable balance) ← state.get_balance(signer)
  if amount ≤ 0 ∨ amount > b_n then
    return ⊥ (invalid amount)
  end

  ▷ Move spendable KOR to staked balance
  state.reduce_balance(signer, amount)
  state.add_stake(signer, amount)
  return success
end
```

2.8.7 Unstake Tokens

Storage nodes invoke the `Unstake-Tokens` procedure to move KOR from their staked balance back to their spendable balance. Withdrawals are programmatically blocked if they would cause the node's stake to fall below the required amount for its current file commitments.

The procedure calculates the node's required stake k_{req} based on its file agreement set and stake amplification factor, validates that the post-withdrawal stake would remain sufficient, then transfers the specified amount from staked to spendable balance.

Algorithm 11: Unstake Tokens

procedure UNSTAKE-TOKENS(state, signer, amount, block_height)
  ▷ Node attempts to move staked KOR to spendable balance
  ▷ Blocked if would result in insufficient stake
  k_n (current stake) ← state.get_stake(signer)

  ▷ Check withdrawal amount validity
  if amount $\leq 0 \vee$ amount $> k_n$ **then**
    return $\perp$ (invalid amount)
  end

  ▷ Calculate required stake for node's file agreements
  $\mathcal{F}_n$ (node file agreements) ← state.get_files_for_node(signer)
  stake_sum ← $\sum_{f \in \mathcal{F}_n}$ state.get_agreement(f).base_stake
  $\lambda_{\text{stake}} \leftarrow 1 + \frac{\lambda_{\text{slash}}}{\ln(2+|\mathcal{F}_n|)}$
  $k_{\text{required}} \leftarrow \text{stake\_sum} \cdot \lambda_{\text{stake}}$
  ▷ Total required stake from economic model [3]

  ▷ Check if withdrawal would violate stake requirement
  k'_n (stake after withdrawal) ← $k_n - \text{amount}$
  if $k'_n < k_{\text{required}}$ **then**
    return $\perp$ (insufficient stake after withdrawal)
  end
  ▷ Withdrawal programmatically blocked to maintain sufficiency

  ▷ Execute withdrawal
  state.reduce_stake(signer, amount)
  state.add_balance(signer, amount)
  return success
end

2.9 Block Processing

The protocol executes deterministic state transitions at the start and end of each block, independent of user transactions. These algorithms are called from the Block Start and Block End procedures defined in the Protocol Flow section.

2.9.1 Challenge Generation

Kontor indexers deterministically derive a set of challenges from each Bitcoin block using the current block hash as the entropy source. For each file selected, one of its storing nodes is randomly chosen to produce a proof.

Algorithm 12: Create Challenge

procedure CREATE-CHALLENGE(node_id, file_id, metadata, block_height, batch_seed, params)
  ▷ Step 1: Compute deterministic challenge identifier
  σ_{batch} ← batch_seed
  ρ ← metadata.root
  d ← trailing_zeros(metadata.padded_len)
  s ← params.num_sectors
  $\mathcal{J} \leftarrow \mathcal{H}_{\text{SHA256}}(\text{TAG}_{\text{challenge\_id}} \parallel \text{block\_height} \parallel \sigma_{\text{batch}} \parallel \text{file\_id} \parallel \rho \parallel d \parallel s \parallel \text{node\_id})$
  ▷ Deterministic ID with domain separation
  ▷ Includes all challenge parameters for uniqueness

  ▷ Step 2: Set challenge parameters and expiration

```
    expiration_block ← block_height + W_proof
    s ← s_chal
    ▷ Must respond within W_proof blocks with s symbol proofs

    ▷ Step 3: Package challenge structure
    C ← {id : J, node_id : node_id, file_id : file_id, metadata : metadata, block_height :
    block_height, expiration_block : expiration_block, num_sectors : s, seed : σ_batch}
    return C
end
```

Algorithm 13: Generate Challenges for Block

```
procedure GENERATE-CHALLENGES-FOR-BLOCK(state, block_height, block_hash)
    ▷ Step 1: Derive deterministic randomness for this block
    H ← block_hash
    info ← KONTOR-CHAL::v1 ‖ block_height
    σ_batch ← HKDF_SHA256(H, info)
    ▷ Current block hash provides unpredictable entropy

    ▷ Step 2: Calculate challenge probability (constant across all files)
    p_f ← C_target / B
    ▷ Challenge probability from global parameters

    ▷ Step 3: Probability-based file selection
    F ← state.get_files(block_height)
    F_selected ← empty list
    for f ∈ F do
        ▷ Derive file-specific random value
        u_f ← H_SHA256(σ_batch ‖ f.id) mod 2^32 / 2^32
        ▷ Uniform u_f ∈ [0, 1)
        if u_f < p_f then
            F_selected.append(f)
        end
    end

    ▷ Step 4: Select one node per challenged file agreement
    ▷ Node n ∈ N_f selected with probability 1/|N_f|
    C_new ← empty list
    for f ∈ F_selected do
        N_f ← state.get_nodes_for_file(f.id)
        if |N_f| > 0 then
            n ← RandomChoice(N_f, σ_batch)
            ▷ Uniform random selection: probability 1/|N_f|
            C ← Create-Challenge(n, f.id, f.metadata, block_height, σ_batch, params)
            C_new.append(C)
        end
    end
    return C_new
end
```

2.9.2 Reward Emissions

Every block, the protocol mints new KOR and distributes it to storage nodes proportionally based on their file agreements and sponsorship arrangements. The total emissions are determined by the dynamic monetary policy, calculated as:

$$\varepsilon = \mathrm{KOR}_{\mathrm{total}} \cdot \frac{\alpha \cdot \mu_0}{B} \tag{10}$$

where the emission rate multiplier α responds to network health:

$$\alpha = \begin{cases} \min\left(\alpha_{\max}, 1 + \frac{\kappa_\alpha}{m}\right) & \text{if } m > 0 \\ \alpha_{\max} & \text{if } m \leq 0 \end{cases} \tag{11}$$

where $m = \overline{R} - n_{\min}$ is the health buffer between the current emission-weighted replication and the minimum threshold. See [3, § Monetary Policy and Emissions] for economic analysis and parameter selection.

Each file receives a share based on its emission weight. The distribution accounts for commission payments between sponsors and entrants.

Edge Cases:

- **Network bootstrap** ($\mathcal{F} = \emptyset$): When no files exist, $\Omega = 1.0$ (genesis bootstrap value) and no emissions are distributed. This initialization prevents division-by-zero in stake calculations for the first file.
- **Abandoned files** ($|\mathcal{N}_f| = 0$): When all nodes have been removed from a file agreement (through slashing or stake insufficiency), the file remains in $\mathcal{F}$ and continues to accrue emissions, but no nodes receive rewards. These emissions are effectively burned (not minted). The file's ω_f remains in the global Ω calculation to maintain deterministic stake calculations for new files.

Algorithm 14: Storage Rewards Distribution

procedure DISTRIBUTE-STORAGE-REWARDS(state, block_height)
  ▷ Called every block during On-Block-End
  ▷ Mints and distributes KOR emissions to storage nodes

  ▷ Step 1: Calculate total block emissions
  $\Omega \leftarrow$ state.get_omega()
  $\text{KOR}_{\text{total}} \leftarrow$ state.get_total_supply()
  $\alpha \leftarrow$ state.get_emission_multiplier()
  $\varepsilon \leftarrow \text{KOR}_{\text{total}} \cdot \frac{\alpha \cdot \mu_0}{B}$
  ▷ Total emissions from monetary policy [3]

  ▷ Step 2: Distribute emissions per file
  $\mathcal{F} \leftarrow$ state.get_active_files()
  for $f \in \mathcal{F}$ **do**
    ▷ Calculate file-specific emissions
    $\mathcal{A} \leftarrow$ state.get_agreement(f)
    $\omega_f \leftarrow \mathcal{A}$.emission_weight
    $\varepsilon_f \leftarrow \varepsilon(t) \cdot \left(\frac{\omega_f}{\Omega}\right)$
    ▷ File's proportional share of total emissions

    ▷ Get nodes storing this file
    $\mathcal{N}_f \leftarrow$ state.get_nodes_for_file(f)
    if $|\mathcal{N}_f| = 0$ **then**
      ▷ File abandoned: emissions not minted (effectively burned)
      ▷ Skip to next file

      ▷ Calculate base per-node reward
      $r_{\text{base}} \leftarrow \frac{\varepsilon_f}{|\mathcal{N}_f|}$

      ▷ Step 3: Distribute to each node with commission adjustments
      for $n \in \mathcal{N}_f$ **do**
        ▷ Initialize commission terms
        $\gamma_{\text{paid}} \leftarrow 0$
        $\gamma_{\text{earned}} \leftarrow 0$

        ▷ Check if node is entrant in active sponsorship for this file
        for $m \in$ state.sponsorships **do**
          if m.file_id $= f \wedge m$.entrant $= n$ **then**
            if block_height $\geq m.t_{\text{start}} \wedge$ block_height $< m.t_{\text{start}} + m.\gamma_{\text{duration}}$ **then**

```
                        γ_paid ← m.γ_rate
                        ▷ Entrant pays commission to sponsor
                    end
                end
            end

            ▷ Check if node is sponsor for entrants in this file
            for m ∈ state.sponsorships do
                if m.file_id = f ∧ m.sponsor = n then
                    if block_height ≥ m.t_start ∧ block_height < m.t_start + m.γ_duration then
                        γ_earned ← γ_earned + m.γ_rate
                        ▷ Sponsor earns commission from entrant
                    end
                end
            end

            ▷ Compute final reward with commission adjustments
            r ← r_base · (1 − γ_paid + γ_earned)
            ▷ Entrant pays out, sponsor earns in

            ▷ Mint and add to node's spendable balance
            state.mint_tokens(r)
            state.add_balance(n, r)
        end
    end
    ▷ End of |N_f| > 0 case
  end
end
```

2.9.3 Challenge Expiration and Slashing

If a storage node either lets a challenge expire or produces an invalid storage proof (), then that storage node's stake k_n is slashed by an amount equal to $\lambda_{\text{slash}} \cdot k_f$, where k_f is the base stake for the file in question and λ_{slash} is a system-wide multiplier. The node is also immediately removed from the file agreement.

A proportion of the slashed funds, β_{slash}, is burned. The remainder is distributed equally among the other storage nodes that are parties to the file agreement that was broken. This disincentivizes a form of collusion in which only one storage node in the agreement actually stores the file and merely transfers the file data to other nodes that have committed to it when the latter are challenged.

If a slash (or any other event) causes a node's total stake k_n to fall below its required stake k_{req}, the protocol automatically triggers a stake-sufficiency-restoration process. The node is gracefully removed from file agreements (with a penalty deducted from stake for each involuntary exit) until its stake is sufficient again. If this is not possible without violating minimum replication on its remaining file agreements, its entire remaining stake is burned, and it is removed from all agreements.

- If $k_n < k_{\text{req}}$ after slashing, the Stake Insufficiency Handling algorithm is automatically triggered.
- The slashed node n is immediately removed from $\mathcal{N}_f$ and file agreement f is removed from $\mathcal{F}_n$.
- Any sponsorship agreements where node n was either an entrant or a sponsor are immediately voided and removed from $\mathcal{M}$.

Algorithm 15: Failure Detection and Slashing

```
procedure PROCESS-FAILED-CHALLENGES(state, block_height)
    ▷ Identify and process all failed challenges
```

```
    ▷ Collect expired challenges (not submitted within W_proof blocks)
    C_expired ← empty list
    for C ∈ state.active_challenges do
        if block_height ≥ C.expiration_block then
            C_expired.append(C)
        end
    end

    ▷ Move expired challenges to failed queue
    for C ∈ C_expired do
        STATE.failed_challenges.add(C.id)
    end

    ▷ Apply penalties to all failed challenges
    for challenge_id ∈ state.failed_challenges do
        C_failed ← state.challenges.get(challenge_id)
        ▷ Find file agreement for challenged file
        A_id ← state.get_agreement_for_file(C_failed.file_id)
        if A_id ≠ ⊥ then
            A ← state.get_agreement(A_id)
            node_id ← C_failed.node_id

            ▷ Calculate slash penalty: k_f × λ_slash
            k_f ← A.base_stake
            penalty ← k_f × λ_slash

            ▷ Burn and distribute penalty
            burn_amount ← penalty × β_slash
            distribute_amount ← penalty × (1 − β_slash)
            STATE.reduce_stake(node_id, penalty)
            STATE.burn(burn_amount)

            ▷ Distribute to remaining nodes in file agreement
            N_f (other nodes) ← A.nodes \ node_id
            if |N_f| > 0 then
                share ← distribute_amount / |N_f|
                for n_i ∈ N_f do
                    STATE.add_stake(n_i, share)
                end
                ▷ No remaining nodes: burn entire distribution
                STATE.burn(distribute_amount)
            end

            ▷ Handle bond transfer if first challenge failure for sponsored join
            file_id ← A.file_id
            for m ∈ state.sponsorships do
                if m.file_id = file_id ∧ m.entrant = node_id ∧ m.first_proof_complete =
                false then
                    ▷ Entrant failed first challenge: void sponsorship retroactively

                    ▷ Transfer bond to sponsor as compensation
                    bond ← state.get_bond_escrow((node_id, file_id))
                    state.add_balance(m.sponsor, bond)
                    state.clear_bond_escrow((node_id, file_id))
                    ▷ Sponsor compensated for fees and bandwidth costs

                    ▷ Void sponsorship - no commission ever paid
                    state.sponsorships.remove(m)
                    ▷ Retroactive void: sponsor earned no commission
                end
            end

            ▷ Remove node from file agreement
```

```
        N_f ← state.get_nodes_for_file(A.file_id)
        N_f.remove(node_id)
        state.set_nodes_for_file(A.file_id, N_f)
        F_n ← state.get_files_for_node(node_id)
        F_n.remove(A.file_id)
        state.set_files_for_node(node_id, F_n)
        A.nodes.remove(node_id)

        ▷ Void other sponsorships involving slashed node
        for m ∈ state.sponsorships do
            if (m.entrant = node_id ∨ m.sponsor = node_id) ∧ m.file_id ≠ file_id then
                state.sponsorships.remove(m)
                ▷ Other sponsorships voided, bonds handled separately
            end
        end

        ▷ Check if stake insufficiency triggered
        k_n (node stake) ← state.get_stake(node_id)
        F_n ← state.get_files_for_node(node_id)
        stake_sum ← Σ_{f∈F_n} state.get_agreement(f).base_stake
        λ_stake ← 1 + λ_slash / ln(2+ |F_n|)
        k_required ← stake_sum · λ_stake
        ▷ Total required stake from economic model [3]
        if k_n < k_required then
            ▷ Trigger stake insufficiency handling
            Handle-Stake-Insufficiency(node_id, state)
        end
    end
    ▷ Clear processed failures
    STATE.failed_challenges.clear()
end
```

2.9.4 Stake Insufficiency Handling

- **Stake Insufficiency Handling:** If a node's stake falls below its requirement (e.g., after being slashed), this automated process restores sufficiency by removing it from file agreements in a pseudo-random order. This operation takes precedence over withdrawal.
  - **Pass 1 (Graceful Exit):** The node is removed from file agreements where its departure is non-critical ($|\mathcal{N}_f| > n_{\text{min}}$). This continues until sufficiency is met. For each involuntary exit from file agreement f, the protocol deducts a penalty from the node's staked balance k_n:
    - Of this penalty $k_f \cdot \lambda_{\text{slash}}$, an amount $\beta_{\text{slash}} \cdot k_f \cdot \lambda_{\text{slash}}$ is burned.
    - The remainder $(1 - \beta_{\text{slash}}) \cdot k_f \cdot \lambda_{\text{slash}}$ is distributed equally among the other storers in $\mathcal{N}_f$.
  - **Pass 2 (Total Forfeiture):** If Pass 1 is insufficient, the node's entire remaining stake k_n is burned, and the node is removed from all remaining file agreements. This ensures the node pays the maximum possible penalty and cannot game the insufficiency mechanism.

Algorithm 16: Stake Insufficiency Handling

```
procedure HANDLE-STAKE-INSUFFICIENCY(node_id, state)
    ▷ Automatic restoration when k_n < k_req after slashing
    ▷ Takes precedence over withdrawal operations
    k_n (node stake) ← state.get_stake(node_id)
    F_n (node file agreements) ← state.get_files_for_node(node_id)
    stake_sum ← Σ_{f∈F_n} state.get_agreement(f).base_stake
    λ_stake ← 1 + λ_slash / ln(2+ |F_n|)
```

```
    k_required ← stake_sum · λ_stake
    ▷ Total required stake from economic model [3]
    ▷ Check if insufficiency exists
    if k_n ≥ k_required then
        return success (no action needed)
    end


    ▷ PASS 1: Graceful Exit from Non-Critical File Agreements
    ▷ Remove from file agreements where |N_f| > n_min
    files_to_remove ← empty list
    for f ∈ F_n do
        ▷ Check if node can be removed without violating minimum
        N_f ← state.get_nodes_for_file(f)
        if |N_f| > n_min then
            files_to_remove.append(f)
        end
    end

    ▷ Shuffle for pseudo-random order
    seed ← H_SHA256(node_id ‖ block_height)
    Shuffle(files_to_remove, seed)
    ▷ Prevents node from predicting removal order to game penalties

    ▷ Remove from file agreements until sufficiency restored
    for f ∈ files_to_remove do
        ▷ Get file agreement and calculate penalty
        A_id ← state.get_agreement_for_file(f)
        A ← state.get_agreement(A_id)
        k_f ← A.base_stake
        penalty ← k_f × λ_slash
        ▷ Same penalty as failed challenge: k_f × λ_slash

        ▷ Apply penalty for involuntary exit
        burn_amount ← penalty × β_slash
        distribute_amount ← penalty × (1 − β_slash)
        STATE.reduce_stake(node_id, penalty)
        k_n ← k_n − penalty
        ▷ Update local stake tracker
        STATE.burn(burn_amount)

        ▷ Distribute to remaining honest storers
        N_f ← state.get_nodes_for_file(f)
        N'_f ← empty set
        for n ∈ N_f do
            if n ≠ node_id then
                N'_f.add(n)
            end
        end
        if |N'_f| > 0 then
            share ← ⌊distribute_amount / |N'_f|⌋
            for n_i ∈ N'_f do
                STATE.add_stake(n_i, share)
            end
            ▷ No other storers: burn entire distribution
            STATE.burn(distribute_amount)
        end

        ▷ Remove node from file agreement
        N_f.remove(node_id)
        state.set_nodes_for_file(f, N_f)
        F_n.remove(f)
        state.set_files_for_node(node_id, F_n)
```

```
        A.nodes.remove(node_id)
        ▷ Node forcibly removed from file agreement

        ▷ Void sponsorships involving removed node for this file
        for m ∈ state.sponsorships do
            if (m.file_id = f) ∧ (m.entrant = node_id ∨ m.sponsor = node_id) then
                state.sponsorships.remove(m)
            end
        end

        ▷ Check if sufficiency restored after this removal
        stake_sum ← ∑_{f∈F_n} state.get_agreement(f).base_stake
        λ_stake ← 1 + λ_slash / ln(2+|F_n|)
        k_required ← stake_sum · λ_stake
        ▷ Recompute required stake with updated file agreement set
        if k_n ≥ k_required then
            return success (sufficiency restored)
        end
        ▷ If sufficient, exit early; otherwise continue removing file agreements
    end


    ▷ PASS 2: Total Forfeiture
    ▷ If Pass 1 insufficient, burn all remaining stake and remove from all file agreements
    if k_n < k_required then
        ▷ Node still insufficient after graceful exits
        ▷ Apply maximum penalty: burn entire remaining stake
        stake_remaining ← state.get_stake(node_id)
        if stake_remaining > 0 then
            STATE.reduce_stake(node_id, stake_remaining)
            STATE.burn(stake_remaining)
        end
        ▷ All stake forfeited to prevent gaming insufficiency mechanism

        ▷ Remove from all remaining file agreements
        F_n ← state.get_files_for_node(node_id)
        ▷ May include file agreements where |N_f| ≤ n_min (critical agreements)
        for f ∈ F_n do
            ▷ Update node membership for this file agreement
            N_f ← state.get_nodes_for_file(f)
            N_f.remove(node_id)
            state.set_nodes_for_file(f, N_f)

            ▷ Remove from file agreement structure
            A_id ← state.get_agreement_for_file(f)
            A ← state.get_agreement(A_id)
            A.nodes.remove(node_id)
            ▷ File agreement may now be under-replicated if |N_f| < n_min, but remains in F and active - never deactivates
        end

        ▷ Clear node's entire file agreement set
        state.set_files_for_node(node_id, ∅)

        ▷ Void all sponsorships involving this node
        for m ∈ state.sponsorships do
            if m.entrant = node_id ∨ m.sponsor = node_id then
                state.sponsorships.remove(m)
            end
        end
```

```
        ▷ Node ejected from protocol with total forfeiture
        return total_forfeiture
    end

    ▷ Should not reach here: either Pass 1 succeeded or Pass 2 executed
    return ⊥ (unexpected state)
end
```

3 Cryptographic System

3.1 Multi-File Proof Aggregation

Storage nodes are frequently challenged on multiple files within the proof submission window (W_{proof} blocks). To minimize Bitcoin transaction fees, the protocol allows nodes to aggregate challenges for multiple files into a single proof. This aggregation is enabled by the File Ledger, a Merkle tree built over the root commitments of all files in the system.

Root Commitment: For each file f with Merkle root ρ_f and tree depth d_f, the protocol computes a root commitment:

$$\text{rc}_f = \mathcal{H}_{\text{Poseidon}}\big(\text{TAG}_{\text{rc}}, \rho_f, d_f\big) \tag{12}$$

The root commitment binds together the file's Merkle root and its tree depth, preventing depth-spoofing attacks where an adversary might try to reuse proofs across files with different tree structures.

File Ledger Construction: The File Ledger is a Merkle tree $\mathcal{L}$ built over the root commitments of all files in the system. Files are ordered deterministically (lexicographically by file identifier), and the ledger tree is constructed from their rc values:

$$\mathcal{L} = \text{Merkle-Tree}\Big(\big[\!\big[\text{rc}_1, \text{rc}_2, ..., \text{rc}_{|\mathcal{F}|}\big]\!\big]\Big) \tag{13}$$

Each file has a canonical ledger index i corresponding to its position in this sorted order. The ledger root $\rho_{\mathcal{L}}$ commits to the set of all files in the system.

Aggregated Proof Structure: When a node is challenged on k files $\{f_1, f_2, ..., f_k\}$, it generates a single proof. The uncompressed IVC witness contains:

- For each challenged file f_j: Merkle path from rc_{f_j} to ledger root $\rho_{\mathcal{L}}$ (size: $O(\log|\mathcal{F}|)$)
- For each challenged symbol in each file: Merkle path from symbol to file root ρ_{f_j} (size: $O(\log n_{\text{total}})$)
- Nova IVC accumulator tracking all verifications (size: grows with number of steps)

Total uncompressed size: $O(k \cdot \log|\mathcal{F}|) + O(k \cdot s_{\text{chal}} \cdot \log n_{\text{total}})$ where s_{chal} is symbols challenged per file.

The compressed proof (after Spartan compression) is constant-size (≈ 10 kB) regardless of k, $|\mathcal{F}|$, or the number of challenged symbols. Nova's IVC folding combined with Spartan's succinct verification enables this compression: the variable-size witness reduces to a constant-size SNARK that proves all symbol verifications and ledger inclusions were performed correctly.

3.1.1 Multi-File Public Inputs

The public input structure differs between single-file and multi-file proofs. For single-file proofs (one challenged file), the public inputs are:

$$\boldsymbol{z}_0 = [\rho, s_0, \sigma, d] \tag{14}$$

where ρ is the file's Merkle root, $s_0 = 0$ is the initial state accumulator, σ is the challenge seed, and d is the tree depth.

For multi-file proofs (multiple challenged files), the public input structure is extended to accommodate per-file parameters and the aggregated ledger. Let k be the number of challenged files. The public inputs are organized as:

$$\boldsymbol{z}_0 = \left[\rho_{\mathcal{L}}, s_0, \boldsymbol{I}_{\text{ledger}}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{Z}_{\text{out}}\right] \tag{15}$$

where:

- $\rho_{\mathcal{L}}$ - Aggregated ledger root (commits to all files in system)
- s_0 - Initial state accumulator (0)
- $\boldsymbol{I}_{\text{ledger}} = [i_1, ..., i_k]$ - Ledger indices for each challenged file (canonical positions in sorted ledger)
- $\boldsymbol{D} = [d_1, ..., d_k]$ - Tree depths for each challenged file
- $\boldsymbol{\Sigma} = [\sigma_1, ..., \sigma_k]$ - Challenge seeds for each file
- $\boldsymbol{Z}_{\text{out}} = [z_1, ..., z_k]$ - Output leaf field elements for each file (final state after processing all symbols)

The circuit verifies that each file's root commitment $\text{rc}_j = \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{rc}}, \rho_j, d_j)}$ is located at ledger index i_j in the tree with root $\rho_{\mathcal{L}}$, then proceeds to verify symbol Merkle paths for each challenged file using the per-file parameters.

This layout enables the circuit to process multiple files in a single IVC proof while maintaining independent verification of each file's challenged symbols.

3.1.2 Proof Structure and Challenge Binding

Storage proofs must cryptographically bind to the specific challenges they answer. The proof structure includes:

$$\begin{aligned} \pi = \big(&\pi_{\text{compressed}}, \text{compressed SNARK} \\ &\textbf{challenge\_ids}, \text{ordered list of challenge identifiers} \\ &\boldsymbol{z}_0, \text{public inputs} \\ &\boldsymbol{z}_s \text{ final state after } s \text{ iterations}\big) \end{aligned} \tag{16}$$

Challenge ID Ordering: The `challenge_ids` list must be ordered consistently with the proof's internal structure. For multi-file proofs, the order of challenge IDs must match the order of files in the public inputs ($\boldsymbol{I}_{\text{ledger}}$, $\boldsymbol{D}$, $\boldsymbol{\Sigma}$). This ordering constraint ensures:

- Verifiers can deterministically reconstruct the exact public inputs
- Each challenge maps to its corresponding file parameters
- Proof substitution attacks are prevented (cannot reorder challenges post-proof)

Verification Binding: When verifying a multi-challenge proof, indexers must:

1. Extract the ordered `challenge_ids` list from the proof structure
2. Retrieve each challenge $\mathcal{C}_j$ by ID from protocol state
3. Verify length: number of challenge IDs must match expected count
4. Verify order: challenge IDs must be in the same order as files in public inputs
5. Reconstruct public inputs from challenges in the verified order
6. Verify the SNARK against these deterministic public inputs

Any mismatch in length or order causes immediate proof rejection. This prevents an adversary from:

- Submitting a valid proof for different challenges (ID mismatch)
- Reordering challenges to exploit different ledger states (order mismatch)
- Claiming to answer more/fewer challenges than proven (length mismatch)

Ledger State Binding: Proofs are cryptographically bound to a specific ledger state through the aggregated ledger root $\rho_{\mathcal{L}}$ in the public inputs. This root must correspond to the ledger state at the challenge block height, not the current state. The binding prevents:

- **Ledger substitution**: Cannot prove against a different file set with a different ledger root

- **Index manipulation**: Cannot claim files are at different ledger indices than their canonical positions
- **State confusion**: Cannot use historical proofs against current challenges or vice versa

The combination of challenge ID ordering and ledger root binding ensures that proofs are uniquely tied to specific challenges at specific protocol states, preventing all forms of proof reuse or substitution attacks.

3.1.3 Ledger Operational Details

The File Ledger $\mathcal{L}$ is a dynamic Merkle tree that grows as new file agreements activate. This section specifies the operational algorithms for ledger maintenance, caching strategies, and versioning semantics.

Ledger Depth: The aggregated ledger depth depends on the padded leaf count. The ledger pads to the next power of two:

$$n'_{\text{ledger}} = \begin{cases} 1 & \text{if } |\mathcal{F}| = 0 \\ \text{next\_power\_of\_two}(|\mathcal{F}|) & \text{otherwise} \end{cases} \tag{17}$$

The depth is then $d_{\text{agg}} = \log_2 n'_{\text{ledger}}$, which equals 0 when $|\mathcal{F}| \in \{0, 1\}$, and $\lceil \log_2 |\mathcal{F}| \rceil$ for $|\mathcal{F}| \geq 2$. The tree remains balanced at all sizes.

Update Algorithm: When a file agreement activates in `Join-Agreement` (reaching $n_{\min}$ nodes), the indexer updates the ledger:

1. **Compute root commitment**: $\text{rc}_{\text{new}} = \mathcal{H}_{\text{Poseidon}}(\text{TAG}_{\text{rc}}, \rho_{\text{new}}, d_{\text{new}})$ using the new file's Merkle root and depth
2. **Insert entry**: Add the file identifier and root commitment to the ledger's ordered map structure
3. **Rebuild tree**: Reconstruct the complete Merkle tree from all root commitments in lexicographic file identifier order, padding to the next power of two with zero elements

The rebuild operation requires $O(|\mathcal{F}|)$ work but occurs only when files activate. Lexicographic ordering ensures deterministic file indices across all indexers: all implementations must sort by file identifier bytes and rebuild the tree in identical order.

Indexer Caching: Indexers maintain the complete ledger tree structure:
- All leaf values (root commitments for each file in $\mathcal{F}$)
- All internal nodes (cached intermediate hashes)
- Sorted mapping from file identifier to ledger index

This full tree enables efficient Merkle path generation for challenge verification. The storage cost is $O(|\mathcal{F}|)$ field elements: for $|\mathcal{F}|$ files padded to n'_{ledger}, the tree contains n'_{ledger} leaves and $n'_{\text{ledger}} - 1$ internal nodes, totaling $\approx 2n'_{\text{ledger}}$ field elements $\approx 64\ |\mathcal{F}|$ bytes. For a network with 1 million files, the ledger tree requires ≈ 64 MB of indexer storage.

Storage Node Caching: Storage nodes do not need to maintain the complete ledger tree. Instead, when generating proofs, nodes obtain:
- The current ledger root $\rho_{\mathcal{L}}$ from recent blockchain state
- Merkle paths for their challenged files from an indexer or by reconstructing locally

Nodes storing $|\mathcal{F}_n|$ files can cache just their subset of root commitments and ledger indices ($\approx 64\ |\mathcal{F}_n|$ bytes). If a node chooses to maintain the full ledger tree locally (for autonomy), the storage cost is identical to indexers.

Historical Ledger Roots: The protocol requires verifying proofs against the ledger root at challenge block height. When a challenge $\mathcal{C}$ is generated at block h, the challenge must record:

$$\rho_{\mathcal{L}}(h) = \text{ledger\_root\_at\_block}(h) \tag{18}$$

This ensures nodes prove against a stable ledger state, preventing race conditions if files activate during the proof window. Indexers must maintain:

- Current complete ledger tree (all leaves and internal nodes)
- Ledger root at each block where a file activated or challenge was issued
- Activation log: which files were in $\mathcal{F}$ at each block height

Historical Reconstruction: To verify a proof against a historical root $\rho_{\mathcal{L}}(h)$, an indexer can:

1. Filter $\mathcal{F}$ to only files active at block h using the activation log
2. Rebuild the tree from those root commitments (in lexicographic order)
3. Verify the rebuilt root matches the recorded $\rho_{\mathcal{L}}(h)$

This reconstruction is $O(|\mathcal{F}(h)|)$ work but only needed for auditing or when the recorded historical root is unavailable. For routine verification, indexers use the recorded historical root directly from the challenge.

Storage Cost: Historical root tracking grows at one entry per file activation. For a network activating 1000 files per day, storage grows at ≈ 40 kB/day (≈ 14 MB/year): 8 bytes block height + 32 bytes root per activation. This is negligible compared to the ledger tree itself (≈ 64 MB for 1M files).

3.2 Proof Generation (Nova IVC)

Algorithm 17: Proof Generation (Nova IVC)

procedure NOVA.PROVE(challenge_id, node_storage, state)
  ▷ Node generates proof for a challenge
  ▷ Step 1: Retrieve challenge and file data
  $\mathcal{C} \leftarrow$ state.get_challenge(challenge_id)
  file_id $\leftarrow \mathcal{C}$.file_id
  M (metadata) $\leftarrow \mathcal{C}$.metadata
  s (sectors to prove) $\leftarrow \mathcal{C}$.num_sectors
  σ (seed) $\leftarrow \mathcal{C}$.seed

  ▷ Step 2: Extract file tree and metadata
  $\mathcal{T}$ (Merkle tree) $\leftarrow$ node_storage.get_tree(file_id)
  ρ (root) $\leftarrow M$.root
  n' (padded leaves) $\leftarrow M$.padded_len
  d (tree depth) $\leftarrow$ trailing_zeros(n')
  ▷ Depth derived from padded length (power of two)

  ▷ Step 3: Generate public parameters for circuit
  ▷ Parameters depend on tree depth and number of sectors
  pp (public params) $\leftarrow \mathcal{G}(d, s)$
  ▷ Nova setup: Pallas/Vesta curve cycle with IPA polynomial commitment

  ▷ Step 4: Build initial public input (single-file case)
  s_0 (initial state) $\leftarrow 0$
  $\boldsymbol{z}_0 \leftarrow [\rho, s_0, \sigma, d]$
  ▷ Single-file public inputs: root, state, seed, depth
  ▷ Multi-file proofs use extended structure: see Section 3.1.1

  ▷ Step 5: Initialize Nova IVC accumulator
  $\Pi_0 \leftarrow$ Nova.Init(pp, $\boldsymbol{z}_0$)
  ▷ Base case with no iterations yet

  ▷ Step 6: Iteratively prove s sectors
  for $i \in [1, s]$ **do**
    ▷ Derive challenge index using domain-separated hash
    $h_i \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge}}, \sigma, s_{i-1})}$
    $c_i \leftarrow$ derive_index_unbiased(h_i, n')
    ▷ Unbiased index in $[0, n')$ via rejection sampling

```
    ▷ Extract witness: challenged leaf and Merkle path
    ℓ_i (leaf) ← T.get_leaf(c_i)
    π_path (siblings) ← T.get_path(c_i)
    β (directions) ← compute_directions(c_i, d)
    w_i ← [ℓ_i, π_path, β]
    ▷ Witness: leaf data, sibling hashes, path directions

    ▷ Execute step circuit: verify path and update state
    z_i ← φ_PoR(z_{i−1}, w_i)
    ▷ Circuit verifies Merkle path and accumulates state

    ▷ Fold this iteration into IVC accumulator
    Π_i ← Nova.Fold(pp, Π_{i−1}, w_i, z_{i−1}, z_i)
    ▷ Incrementally verifiable: proves correctness of step i

    ▷ Update state for next iteration
    s_i ← z_{i[1]}
  end

  ▷ Step 7: Compress IVC accumulator into succinct SNARK
  π_compressed ← Spartan.Compress(pp, Π_s)
  ▷ Final proof: succinct, constant-size, verifiable on-chain

  ▷ Step 8: Package proof with challenge identifiers
  proof ← {challenge_id : challenge_id, compressed_snark : π_compressed, public_inputs :
  z_0, final_state : z_s}
  return proof
end
```

3.3 PoR Step Circuit φ_{PoR}

The PoR step circuit φ_{PoR} is executed within the Nova IVC fold operation. It verifies a single symbol's Merkle path and updates the state accumulator.

Algorithm 18: PoR Step Circuit

```
procedure φ_PoR(z_∈, w)
  ▷ Circuit executed inside Nova fold to verify one symbol
  ▷ Step 1: Parse public inputs from previous state
  ρ (file root) ← z_{∈[0]}
  s (state accumulator) ← z_{∈[1]}
  σ (random seed) ← z_{∈[2]}
  d (tree depth) ← z_{∈[3]}
  ▷ State carries: root commitment, accumulator, randomness, metadata

  ▷ Step 2: Parse witness (private input from prover)
  ℓ (challenged leaf) ← w[0]
  π_siblings (path) ← w[1]
  β (directions) ← w[2]
  ▷ Witness: leaf data, d sibling hashes, direction bits

  ▷ Step 3: Verify Merkle path
  ▷ Recompute root from leaf by hashing up the tree
  current ← H_Poseidon(ℓ)
  for j ∈ [0, d) do
    ▷ At level j: hash with sibling based on direction
    if β[j] = 0 then
      current ← H_Poseidon(current, π_siblings[j])
      ▷ Current node is left child
      current ← H_Poseidon(π_siblings[j], current)
      ▷ Current node is right child
```

```
    end
  end
  ρ_computed ← current

  ▷ Step 4: Assert root matches expected value
  ASSERT(ρ_computed = ρ)
  ▷ Proof fails if computed root doesn't match public input

  ▷ Step 5: Update state accumulator
  s' ← H_Poseidon(s,ℓ)
  ▷ Accumulate challenged leaf into running state

  ▷ Step 6: Build output state for next iteration
  z_out ← [ρ, s', σ, d]
  ▷ Pass root, updated state, seed, depth to next fold
  return z_out
end
```

Circuit Properties:

- **Determinism:** All computations are deterministic given public/private inputs
- **Succinctness:** Circuit size is $O(d)$ where d is tree depth
- **Completeness:** Honest prover with valid data always produces accepting proof
- **Soundness:** Prover without data cannot forge valid Merkle paths

3.4 Proof Verification (Nova IVC)

Algorithm 19: Proof Verification (Nova IVC)

```
procedure NOVA.VERIFY(challenge, proof, state)
  ▷ On-chain verification of storage proof
  ▷ Step 1: Retrieve challenge and validate proof structure
  C ← challenge
  if C = ⊥ then
    return ⊥ (challenge not found)
  end
  if proof.challenge_id ≠ C.id then
    return ⊥ (challenge ID mismatch)
  end

  ▷ Step 2: Extract challenge parameters
  M (metadata) ← C.metadata
  ρ (expected root) ← M.root
  s (num sectors) ← C.num_sectors
  σ (seed) ← C.seed
  n' (padded leaves) ← M.padded_len
  d (tree depth) ← trailing_zeros(n')
  ▷ Depth derived from padded length (power of two)

  ▷ Step 3: Reconstruct public inputs
  s_0 (initial state) ← 0
  z_0 ← [ρ, s_0, σ, d]
  ▷ Must match prover's initial public inputs
  if proof.public_inputs ≠ z_0 then
    return ⊥ (public input mismatch)
  end

  ▷ Step 4: Generate verification parameters
  pp (public params) ← G(d, s)
  ▷ Same parameters as prover: depth d, s sectors

  ▷ Step 5: Extract compressed proof
  π_compressed ← proof.compressed_snark
```

```
    ▷ Step 6: Verify SNARK proves s IVC iterations
    ▷ Verifies: prover executed φ_PoR circuit s times
    valid ← Spartan.Verify(pp, π_compressed, z_0, s)
    ▷ Spartan verification: O(log^2 s) time, constant proof size
    if ¬ valid then
        return ⊥ (proof verification failed)
    end

    ▷ Step 7: Validate final state consistency
    z_s ← proof.final_state
    ρ_final ← z_s[0]
    if ρ_final ≠ ρ then
        return ⊥ (final root mismatch)
    end
    ▷ Root must remain unchanged throughout all iterations

    ▷ All checks passed
    return valid
end
```

3.5 Cryptographic Security Properties

This section analyzes the security properties of the Nova-based proof-of-retrievability system. The protocol's security relies on standard cryptographic hardness assumptions and the soundness properties of the underlying SNARK construction.

3.5.1 Security Model

The proof system provides the following security guarantees in the context of proof-of-retrievability:

Soundness (Proof Unforgeability): A computationally bounded adversary who does not possess the challenged file data cannot generate a valid proof that passes verification, except with negligible probability. Formally, for a file with Merkle root ρ and challenge requiring sectors $\{c_1, ..., c_s\}$:

$$\Pr[\pi \leftarrow \mathcal{A}(\rho, c_1, ..., c_s) : \text{Verify}(\pi, \rho, c_1, ..., c_s) = \text{valid}] \leq \text{negl}(\lambda) \tag{19}$$

where $\mathcal{A}$ is any probabilistic polynomial-time adversary without access to the challenged sectors, and λ is the security parameter.

Completeness (Honest Prover Success): An honest prover possessing the complete file data can always generate a valid proof that passes verification, provided the prover follows the protocol correctly and the underlying cryptographic primitives function as specified. This guarantee holds deterministically, not probabilistically.

Public Verifiability: Proofs can be verified by any party possessing only the public parameters, the Merkle root commitment, and the challenge parameters. Verifiers need not possess the file data, trust the prover, or interact with the prover beyond receiving the proof.

3.5.2 Computational Assumptions

The protocol's security depends on the following computational hardness assumptions:

Discrete Logarithm Problem: The security of the Nova proof system relies on the hardness of the discrete logarithm problem over the Pallas and Vesta elliptic curves. These curves provide approximately 128 bits of security under current best-known attacks.

Collision Resistance: The protocol assumes collision resistance for:

- SHA-256: Used for file identifiers and challenge IDs (256-bit output, 128-bit collision resistance)
- Poseidon: Used for Merkle tree construction and in-circuit hashing (targeting 128-bit security)

Collision resistance ensures that:

- File identifiers uniquely identify files (no two files have the same ID)
- Challenge IDs uniquely identify challenges (no duplicate challenges)
- Merkle roots bind to unique file contents (no two files have the same root)

Fiat-Shamir Heuristic: The Nova and Spartan proof systems use the Fiat-Shamir transformation to convert interactive protocols into non-interactive proofs. Security relies on modeling the hash function as a random oracle. While the random oracle model is a strong assumption, it is standard in SNARK constructions and has no known practical attacks when instantiated with strong hash functions.

Transparent Setup: Unlike SNARKs such as Groth16 or PLONK[5], Nova and Spartan do not require a trusted setup ceremony. The public parameters can be generated by anyone deterministically from the circuit structure without relying on secret randomness that must be destroyed. This eliminates the need for multi-party computation ceremonies, removes the risk of toxic waste compromise, and improves the protocol's decentralization properties. Any party can independently generate or verify the public parameters for a given circuit shape.

Quantum Resistance: The protocol is not resistant to quantum computers. The security of elliptic curve cryptography (Pallas/Vesta curves) and the discrete logarithm problem would be broken by a sufficiently large quantum computer running Shor's algorithm. This limitation is shared with essentially all contemporary SNARK systems and blockchain protocols. Should practical quantum computers emerge, the protocol would require migration to post-quantum cryptographic primitives, though no such primitives currently offer comparable performance for SNARK applications.

3.5.3 Attack Resistance

Proof Forgery Attacks: An adversary who does not possess the challenged sectors must break either:

1. The soundness of the Nova/Spartan SNARK (computationally infeasible under discrete log assumption)
2. The collision resistance of Poseidon (computationally infeasible for 128-bit security)
3. The binding property of Merkle trees (follows from collision resistance)

The multi-layered cryptographic construction ensures that breaking any single component is insufficient; the adversary must break multiple independent hardness assumptions simultaneously.

Grinding Attacks on Challenge Selection: Miners could theoretically attempt to grind block hashes to influence challenge selection. However, this attack is economically irrational: the cost of discarding a valid block (forfeiting block rewards and transaction fees, worth hundreds of thousands of dollars) vastly exceeds any benefit from biasing which storage nodes are challenged. The protocol uses the current block hash directly as the entropy source, as the economic disincentive against grinding is overwhelming.

Malleability Attacks: Each challenge has a unique, deterministic identifier computed via domain-separated hashing. The inclusion of the challenge ID, file root, tree depth, and node ID in the hash input prevents an adversary from:

- Reusing a proof for a different challenge (different challenge ID)
- Reusing a proof for a different file (different root)
- Submitting another node's proof (different node ID)

Replay Attacks: Challenge IDs include block height, ensuring challenges from different blocks are distinguishable. The protocol tracks verified and failed challenges, preventing an adversary from resubmitting old proofs for new challenges.

Randomness Quality: Challenge generation relies on the quality of the randomness beacon. The protocol uses Bitcoin[6] block hashes as the entropy source, which inherit the security properties of Bitcoin's proof-of-work consensus:

- High entropy: Block hashes have 256 bits of entropy from mining randomness

- Unpredictable: Cannot be predicted before mining completes
- Economically unbiasable: Miner grinding attacks are economically irrational (block rewards >> challenge manipulation value)
- Independent: Each block's hash is independent of challenges

The HKDF expansion using domain-separated context information ensures that challenge randomness for different blocks and different files is computationally independent, preventing correlation attacks where an adversary might try to predict multiple challenges simultaneously.

3.5.4 Public Parameter Binding

Public parameters in Nova/Spartan proof systems are specific to the circuit shape (number and structure of constraints). The Kontor protocol requires different parameters for different configurations:

Parameter Determinism: For a given circuit shape (files per step, file tree depth, aggregated tree depth), the public parameters are deterministically generated. Any party can independently compute identical parameters from the circuit structure. Parameters are identified by the shape tuple: $(k, d_{\text{file}}, d_{\text{agg}})$ where k is files per step, d_{file} is the file tree depth, and d_{agg} is the aggregated ledger depth.

Parameter Flexibility: Different files may have different tree depths (different file sizes yield different symbol counts). The protocol supports this by:
- Allowing multiple parameter sets for different depths
- Using padding to standardize depths within a batch (all challenged files in single proof must have matching depths)
- Caching frequently used parameter sets to amortize generation costs

Generation Cost: Public parameter generation is computationally expensive (can take minutes for complex circuits) but only needs to be performed once per shape. The reference implementation includes parameter caching to avoid regeneration. The deterministic generation ensures all implementations compute identical parameters for identical shapes, maintaining consensus.

3.5.5 Concrete Security Parameters

Security Level: The protocol targets 128-bit security:
- Pallas/Vesta curves: 126-bit security (254-bit prime fields)
- SHA-256: 128-bit collision resistance (256-bit output)
- Poseidon: Configured for 128-bit security over Pallas field

This security level is sufficient for the protocol's threat model, where attacks on the cryptographic primitives are significantly more expensive than the economic value of individual files.

Field Sizes:
- Pallas scalar field: $p = 2^{254} + 45560315531419706090280762371685220353$
- Vesta scalar field: $q = 2^{254} + 45560315531506369815346746415080538113$
- Field element encoding: 31 bytes maximum (safe for 255-bit fields)

Proof Size: Compressed proofs are approximately 10 kB regardless of the number of challenged files or sectors. This constant size is a consequence of Spartan's succinct verification and enables efficient on-chain verification.

Verification Time: Proof verification requires $O(\log^2 s)$ time where s is the number of IVC steps (sectors challenged). For typical challenges ($s \approx 100$), verification completes in approximately 50 milliseconds on modern hardware, making on-chain verification by indexers practical.

4 Appendix

4.1 Notation and Formal Definitions

We use the following notation throughout this specification:

Basic Notation:

- $\|$ denotes concatenation of byte strings
- $[n]$ denotes the set $\{0, 1, ..., n-1\}$
- $\boldsymbol{x}$ denotes a vector or array
- $|\boldsymbol{x}|$ denotes the length of vector $\boldsymbol{x}$ or cardinality of set $\boldsymbol{x}$
- $\perp$ denotes an error or null value
- $\emptyset$ denotes the empty set
- $\subseteq$ denotes subset or equal to

Hash Functions:

- $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\psi$ denotes a cryptographic hash function with output length ψ bits
- $\mathcal{H}_{\text{SHA256}}$ denotes SHA-256 hash function with $\psi = 256$
- $\mathcal{H}_{\text{Poseidon}}$ denotes the Poseidon hash function over the Pallas/Vesta curves

Structured Objects (Calligraphic letters):

- $\mathcal{A}$ denotes a storage Agreement
- $\mathcal{C}$ denotes a Challenge
- $\mathcal{E}$ denotes an Erasure coding configuration
- $\mathcal{F}$ denotes a File or set of files (context-dependent)
- $\mathcal{G}$ denotes the Nova setup (generator) function
- $\mathcal{I}$ denotes a set of challenge identifiers or ledger indices
- $\mathcal{L}$ denotes the File Ledger (Merkle tree over file root commitments)
- $\mathcal{M}$ denotes the set of active sponsorship agreements
- $\mathcal{N}$ denotes the set of storage Nodes
- $\mathcal{O}$ denotes the set of active sponsorship Offers
- $\mathcal{P}$ denotes a Procedure call (in transaction processing)
- $\mathcal{S}$ denotes circuit Shape (preprocessed circuit parameters)
- $\mathcal{T}$ denotes a Merkle Tree or Transaction (context-dependent)
- $\mathcal{U}$ denotes the set of Users
- $\mathcal{W}$ denotes the Witness extraction function

Greek Letters (Proofs, Seeds, Parameters):

- π denotes a cryptographic proof
- ρ denotes a Merkle root (file-level or ledger-level, distinguished by context or subscript)
- rc_f denotes a root commitment for file f (binds Merkle root and tree depth)
- σ denotes a random seed
- φ_{PoR} denotes the PoR step circuit function
- Π denotes a Nova IVC accumulator
- θ denotes expected number of challenges
- ψ denotes cryptographic security parameter (in bits, used in hash function definitions)
- λ_{slash} denotes slash penalty multiplier (dimensionless, protocol parameter)
- λ_{stake} denotes dynamic stake amplification factor (dimensionless, computed per-node)
- τ denotes symbol size in bytes (fixed at 31 for field element compatibility)
- ω_f denotes file emission weight (dimensionless, immutable per file)
- Ω denotes total network emission weight (dimensionless, single mutable global)
- $\varepsilon_f(t)$ denotes file-specific KOR emissions per block (used in economic functions)
- γ_{rate} denotes sponsorship commission rate (fraction)
- γ_{duration} denotes sponsorship duration (blocks)
- Δh denotes a difference in block heights

Data Structures and Vectors:

- $\boldsymbol{S}$ denotes vector of sectors (data + parity) from erasure coding
- $\boldsymbol{L}$ denotes vector of Merkle tree leaves (field elements)

- $\boldsymbol{D}$ denotes tree depths vector
- $\boldsymbol{I}$ denotes ledger indices vector
- $\boldsymbol{z}$ denotes public input vector (Nova IVC)
- $\boldsymbol{w}$ denotes witness vector (private input to circuit)
- M denotes metadata structure
- F denotes a file structure
- F_{prep} denotes a prepared file structure
- $\mathcal{F}$ denotes the set of active file agreements
- $\mathcal{F}_n$ denotes the set of file agreements stored by node n
- $\mathcal{N}_f$ denotes the set of nodes storing file agreement f

Variables and Indices:

- `block_height` denotes block height
- n denotes temporary variable for counts
- n' denotes padded count (power of 2)
- N denotes number of file slots in a circuit
- s denotes number of sectors or state variable (context-dependent)
- s_f^{bytes} denotes file size in bytes
- s_f denotes total sectors in file f: $s_f = n_{\text{data}} + n_{\text{parity}}$
- $s_{\text{chal}} = 100$ denotes protocol parameter for sectors sampled per challenge
- s'_{chal} denotes actual sectors challenged: $\min(s_{\text{chal}}, s_f)$
- r denotes reward value
- p denotes price or probability (with subscripts for specificity)
- i, j denotes loop indices
- d denotes tree depth
- u denotes a random value
- ℓ denotes a Merkle tree leaf (field element)
- ν denotes data loss fraction (detection threshold)
- z denotes Z-score for statistical tests

Protocol Parameters:

- W_{proof} denotes proof window (blocks to respond to challenge)
- W_{offer} denotes sponsorship offer validity window (blocks before expiration)
- β_{bond} denotes sponsorship bond amount (in KOR, protects sponsor from griefing)
- B denotes expected Bitcoin blocks per year
- C_{target} denotes target annual challenges per file
- s_{chal} denotes sectors per challenge
- λ_{slash} denotes slash penalty rate
- β_{slash} denotes burn fraction of slash penalty
- $n_{\min}$ denotes minimum nodes required to activate agreement
- c_{stake} denotes base capital parameter (in KOR) for stake calculations
- F_{scale} denotes network size normalization factor for stake scaling
- χ_{fee} denotes ratio of user fee to per-node base stake
- k_n denotes stake balance of node n
- b_n denotes spendable balance of node n
- n_{symbols} denotes number of data symbols (31-byte units from file): $n_{\text{symbols}} = \left\lceil \frac{s_f^{\text{bytes}}}{31} \right\rceil$
- $n_{\text{codewords}}$ denotes number of RS codewords: $n_{\text{codewords}} = \left\lceil \frac{n_{\text{symbols}}}{231} \right\rceil$
- n_{total} denotes total symbols including parity: $n_{\text{total}} = n_{\text{codewords}} \times 255$
- Symbol size: 31 bytes (fixed, equals Pallas field element size)
- Codeword structure: 231 data symbols + 24 parity symbols = 255 total (GF(2^8) limit)
- $r_{\text{overhead}} = 0.10$ denotes parity overhead per codeword (10%)
- $\tau = 31$ denotes symbol size in bytes (equals field element encoding size)

Economic Functions (defined in [3]):

These functions must be computed at runtime using current state values:

- $\omega_f = \frac{\ln\left(s_f^{\text{bytes}}\right)}{\ln(1+\text{rank}_f)}$ - file emission weight (dimensionless)
- Ω - total network emission weight (single global value = sum of all ω_f for active files in $\mathcal{F}$)
- $k_f = \left(\frac{\omega_f}{\Omega}\right) \cdot c_{\text{stake}} \cdot \ln\left(1 + |\mathcal{F}_{\frac{\downarrow}{F_{\text{scale}}}}\right)$ - per-node base stake for file f at creation (KOR)

  Computed using current global Ω and $|\mathcal{F}|$ BEFORE adding this file
- $\upsilon_f = \chi_{\text{fee}} \cdot k_f$ - file storage fee paid by user at creation (KOR, burned)
- $\varphi_{\text{leave}}(f,t) = k_f \cdot \left(\frac{n_{\min}}{|\mathcal{N}_f|}\right)^2$ - voluntary leave fee (KOR, burned)

  Depends on: file's k_f, current replication $|\mathcal{N}_f|$ at time t
- $\lambda_{\text{stake}}(n) = 1 + \frac{\lambda_{\text{slash}}}{\ln(2+|\mathcal{F}_n|)}$ - dynamic stake factor for node n

  Depends on: node's file count $|\mathcal{F}_n|$
- $k_{\text{req}}(n,t) = \left(\sum_{f \in \mathcal{F}_n} k_f\right) \cdot \lambda_{\text{stake}}(n)$ - total required stake for node n

  Depends on: node's file set $\mathcal{F}_n$, each file's k_f, node's λ_{stake}

  Predictive vs Reactive Checks: When validating if a node CAN perform an action (join/unstake), compute k_{req} using the PROJECTED state after the action. When checking if a node's current state is valid (after slashing/involuntary exit), use the CURRENT file set $\mathcal{F}_n$
- $r_{\text{storage}}(n,f,t)$ - storage reward per block for node n storing file f (in KOR)

$$r_{\text{storage}}(n,f,t) = \left(\varepsilon_f \frac{t}{|\mathcal{N}_f|}\right) \times \left(1 - \gamma_{\text{paid}}(n,f,t) + \gamma_{\text{earned}}(n,f,t)\right) \tag{20}$$

  where:
  - $\varepsilon_f \frac{t}{|\mathcal{N}_f|}$ is the base per-node reward for file f
  - $\gamma_{\text{paid}}(n,f,t)$ is the commission rate node n pays if it is an entrant for file f (0 if no active sponsorship)
  - $\gamma_{\text{earned}}(n,f,t)$ is the sum of commission rates node n earns as a sponsor for file f (sum over all entrants)

  Commission only applies to the sponsored file f; other files stored by n earn full rewards

4.2 Parameter Selection

This section specifies the consensus-critical parameters that all conforming implementations must use to ensure network-wide consistency. The `Kontor-Crypto` reference implementation accepts many of these as configurable parameters for testing purposes, but production deployments must use the values specified here.

For economic modeling parameters and comprehensive analysis, see [3, § Protocol Parameters].

4.2.1 File Preparation Parameters

Symbol and Erasure Coding Parameters:

- Symbol size: 31 bytes (Pallas field element constraint)
- Data symbols per codeword: 231
- Parity symbols per codeword: 24 (10% overhead)
- Total symbols per codeword: 255 (GF(2^8) maximum)

Derived formulas:

- $n_{\text{symbols}} = \left\lceil \frac{s_f^{\text{bytes}}}{31} \right\rceil$ - data symbols from file
- $n_{\text{codewords}} = \left\lceil \frac{n_{\text{symbols}}}{231} \right\rceil$ - RS codewords needed

- $n_{\text{total}} = n_{\text{codewords}} \times 255$ - total symbols including parity

Rationale: The 31-byte symbol size enables direct encoding to Pallas field elements (255 bits) with no hashing, ensuring proof-of-retrievability. Multi-codeword structure handles arbitrary file sizes within the GF(2^8) symbol limit. Each codeword provides independent fault tolerance with graceful degradation for large files.

Representative configurations for various file sizes:

File Size	n_{symbols}	Codewords	n_{total}	d	C_{IVC}
10 KB	323	2	510	9	900
100 KB	3,226	14	3,570	12	1,200
1 MB	33,826	147	37,485	16	1,600
10 MB	338,251	1,465	373,815	19	1,900
100 MB	3,382,504	14,643	3,733,965	22	2,200

Table 1: Representative configurations with 31-byte symbols and multi-codeword Reed-Solomon over GF(2^8). Each codeword encodes 231 data symbols with 24 parity symbols (255 total). IVC cost is $C_{\text{IVC}} = 100 \times d$. Tree depth scales logarithmically with total symbols (including parity from all codewords).

Field Element Encoding:

- $\tau = 31$ - Symbol size in bytes (equals field element size)

Rationale: Maximum safe encoding size for the 255-bit Pallas scalar field, ensuring all 31-byte symbols map to valid field elements without overflow.

4.2.2 Challenge Parameters

Challenge Frequency:

- $C_{\text{target}} = 12$ - Target annual challenges per file
- $B = 52,560$ - Expected Bitcoin blocks per year
- Derived: $p_f = \frac{C_{\text{target}}}{B} \approx 0.000228$ per block

Rationale: 12 annual challenges provide strong security guarantees (>99.99% annual detection of complete data loss) while keeping proving costs manageable. See [3] for detection probability analysis.

Challenge Sampling:

- $s_{\text{chal}} = 100$ - Symbols sampled per challenge
- Actual: $s'_{\text{chal}} = \min(s_{\text{chal}}, n_{\text{total}})$ for small files

Rationale: 100 symbols provides >99.99% detection probability for 10% data loss while capping proving costs. Files smaller than 100 symbols are fully challenged.

Proof Window:

- $W_{\text{proof}} = 2016$ - Blocks to respond to challenge (approximately 2 weeks)

Rationale: Two-week window allows nodes to aggregate multiple challenges into single proofs, minimizing Bitcoin transaction fees. Also provides operational buffer for node maintenance and network issues.

4.2.3 Sponsorship Parameters

Offer Expiration:

- W_{offer} - Blocks before sponsorship offer expires (recommended: 144 blocks ≈ 1 day)

Rationale: Short expiration limits entrant waiting time if sponsor ghosts while giving reasonable time for data transfer completion.

Bond Amount:

- Recommended: $\beta_{\text{bond}} \cdot \xi_{\text{KOR/USD}} \geq c^{\text{offer}}_{\text{BTC}} \cdot \xi_{\text{BTC/USD}} + c^{\text{USD}}_{\text{transfer}}$

- Typical: $\beta_{\text{bond}} \approx 3$ KOR (covers ~$0.60 in sponsor costs)

Rationale: Bond must fully compensate sponsor for Bitcoin fees and bandwidth costs to prevent profitable griefing attacks. See Security Analysis for attack cost analysis.

4.2.4 File Size Constraints

Limits:

- $s_{\min} = 10$ KB - Minimum file size
- $s_{\max} = 100$ MB - Maximum file size

Rationale: Minimum prevents spam and ensures reasonable proving costs relative to storage value. Maximum is determined by practical constraints (tree depth, memory requirements, proving time) rather than fundamental protocol limitations. With 31-byte sectors, a 100 MB file requires depth 22, which remains practical for proof generation and verification.

4.2.5 Domain Separation

Tag Strings: All domain tags must use these exact context strings:

- "KONTOR::CHALLENGE_ID::v1" - for challenge ID computation
- "KONTOR::CHALLENGE::v1" - for challenge index derivation
- "KONTOR::CHALLENGE_PER_FILE::v1" - for multi-file mixing
- "KONTOR::NODE::v1" - for internal Merkle nodes
- "KONTOR::LEAF::v1" - for leaf hashing
- "KONTOR::RC::v1" - for root commitments

Rationale: Domain separation prevents cross-context hash collisions and makes protocol upgrades explicit through version suffixes. All implementations must use identical tag strings to ensure consensus.

4.3 Cryptographic Primitives

Definition 1 (Collision-Resistant Hash Function). A function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\psi$ is collision-resistant if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability

$$\Pr[(x, x') \leftarrow \mathcal{A}(1^\psi) : x \neq x' \wedge \mathcal{H}(x) = \mathcal{H}(x')] \tag{21}$$

is negligible in ψ.

Definition 2 (Poseidon Hash Function). Poseidon[7] is an algebraic hash function $\mathcal{H}_{\text{Poseidon}} : \mathbb{F}_p^t \to \mathbb{F}_p$ designed for arithmetic circuits over prime field $\mathbb{F}_p$. For this protocol, we use Poseidon over the Pallas and Vesta curve cycle with field moduli $p_{\text{Pallas}} = 2^{254} + 45560315531419706090280762371685220353$ and $q_{\text{Vesta}} = 2^{254} + 45560315531506369815346746415080538113$.

Definition 3 (HKDF - HMAC-based Key Derivation Function). HKDF is a key derivation function specified in RFC 5869 that expands a source of entropy into cryptographically strong pseudorandom output. The protocol uses $\text{HKDF}_{\text{SHA256}}$ with the following signature:

$$\text{HKDF}_{\text{SHA256}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^\psi \tag{22}$$

where the first input is the initial keying material (IKM), the second is optional context information, and ψ is the desired output length in bits. For challenge generation, the protocol uses IKM = current block hash and info = domain separator concatenated with block height.

Definition 3a (Unbiased Index Derivation). The function derive_index_unbiased : $\mathbb{F}_p \times \mathbb{N} \to \mathbb{N}$ maps a field element to an unbiased index in the range $[0, n)$ for arbitrary n:

- If n is a power of two: extract the low $\log_2 n$ bits (exact, efficient)
- Otherwise: use rejection sampling - extract bits for next power of two, reject if $\geq n$, rehash with counter until valid

This ensures uniform distribution over $[0, n)$ without modulo bias. The rehashing uses domain-separated Poseidon: $h_{i+1} = \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge}}, h_i, i)}$.

Definition 3b (Domain Separation Tags). The protocol uses domain separation to prevent cross-protocol and cross-context hash collisions. Each hash operation includes a unique tag (field element) derived from a context string. The following tags are used throughout the protocol:

- $\text{TAG}_{\text{challenge\_id}}$ - SHA-256 domain tag for challenge identifier computation
- $\text{TAG}_{\text{challenge}}$ - Poseidon tag for challenge index derivation and rejection sampling
- $\text{TAG}_{\text{challenge\_per\_file}}$ - Poseidon tag for combining challenge with file index in multi-file proofs
- TAG_{node} - Poseidon tag for hashing internal Merkle tree nodes
- TAG_{leaf} - Poseidon tag for leaf encoding (symbol-to-field-element conversion)
- TAG_{rc} - Poseidon tag for root commitment computation (binds file root and depth)

Domain tags are computed as $\text{TAG}_{\text{context}} = \mathcal{H}_{\text{Poseidon(context string)}}$ where the context string uniquely identifies the operation. This prevents an adversary from constructing valid proofs by reusing hash outputs from different contexts.

Definition 4 (Merkle Tree). A Merkle tree[8] $\mathcal{T}$ over leaves $\boldsymbol{L} = [\ell_0, ..., \ell_{n-1}]$ with hash function $\mathcal{H}$ is a binary tree where:

- Tree has depth $d = \lceil \log_2 n' \rceil$ where $n' \geq n$ is the padded number of leaves (power of 2)
- Each leaf node contains $\mathcal{H}(\ell_i)$
- Each internal node at position (i, j) contains $\mathcal{H}(\text{left} \parallel \text{right})$ where "left" and "right" are its children
- The root $\rho = \mathcal{T}.\text{root}$ commits to all leaves

A Merkle proof π_i for leaf ℓ_i is a path from leaf to root consisting of sibling hashes at each level.

Definition 5 (Reed-Solomon Erasure Code). A Reed-Solomon code[9] over GF(2^8) with 31-byte symbols satisfies:

- Encodes data $\boldsymbol{D} = \left[d_0, ..., d_{n_{\text{data}}-1}\right]$ to codeword $\boldsymbol{C} = \left[c_0, ..., c_{n_{\text{data}}+n_{\text{parity}}-1}\right]$
- Any n_{data} symbols of $\boldsymbol{C}$ suffice to reconstruct $\boldsymbol{D}$
- Tolerates up to n_{parity} symbol erasures or $\left\lfloor \frac{n_{\text{parity}}}{2} \right\rfloor$ symbol errors

In the Kontor protocol, each symbol is a fixed 31-byte unit. The encoding operates at symbol granularity: a codeword of 231 data symbols yields 24 parity symbols (10% overhead), all 31 bytes each.

4.4 Related Work

The Kontor storage protocol builds on foundational work in proofs of retrievability[10], [11] and draws inspiration from several decentralized storage systems, each with distinct design trade-offs.

Decentralized Storage Networks: Filecoin[12] uses proof-of-spacetime and proof-of-replication to incentivize storage providers through a marketplace model with renewable storage deals. Arweave[13] implements a "blockweave" structure with a one-time payment model for permanent storage, though its economic sustainability depends on decreasing storage costs over time. Storj[14] uses erasure coding and reputation systems but relies on trusted auditing rather than zero-knowledge proofs. IPFS[15] provides content-addressed storage but lacks native economic incentives, leading to data availability challenges[16] that motivate the need for incentivized permanence guarantees.

Cryptographic Foundations: The protocol leverages recursive proof composition via Nova[1] folding schemes, enabling efficient aggregation of multiple storage proofs into constant-size SNARKs. The implementation uses the Poseidon[7] hash function optimized for arithmetic circuits, Merkle trees[8] for cryptographic commitments, and Reed-Solomon[9],

[17] erasure coding for fault tolerance. The proof system is built on Spartan[18] for transparent setup without trusted ceremonies.

Design Differentiators: Unlike Filecoin's renewable deals or Arweave's economic speculation, Kontor provides perpetual storage guarantees through continuous emissions tied to the broader smart contract economy. The use of Bitcoin block hashes[6] as an unbiasable randomness beacon eliminates the need for on-chain random number generation. The pooled stake model and dynamic challenge selection create strong incentives for honest storage while maintaining verification efficiency through recursive proof aggregation.

5 Bibliography

[1] Abhiram Kothapalli and Srinath Setty, "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes," 2021. [Online]. Available: https://eprint.iacr.org/2021/370

[2] Microsoft, *Arecibo*. (2024). GitHub. [Online]. Available: https://github.com/microsoft/arecibo

[3] Adam Krellenstein, "Kontor Economic Model," Oct. 2025. [Online]. Available: https://raw.githubusercontent.com/KontorProtocol/Documentation/main/img/2025-10-07_kontor-economic-model.pdf

[4] Henning Pagnia and Felix C Gärtner, "On the Impossibility of Fair Exchange Without a Trusted Third Party„" *Darmstadt University of Technology Technical Report,*.

[5] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge," 2019. [Online]. Available: https://eprint.iacr.org/2019/953

[6] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[7] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems," 2019. [Online]. Available: https://eprint.iacr.org/2019/458

[8] Ralph C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," 1988.

[9] Irving S. Reed and Gustave Solomon, "Polynomial Codes Over Certain Finite Fields," 1960.

[10] Ari Juels and Burton S. Kaliski Jr, "PORs: Proofs of Retrievability for Large Files," 2007.

[11] Hovav Shacham and Brent Waters, "Compact Proofs of Retrievability„" *Springer*.

[12] Protocol Labs, "Filecoin: A Decentralized Storage Network," July 19, 2017. [Online]. Available: https://filecoin.io/

[13] Sam Williams, Viktor Diordiiev, Lev Berman, India Raybould, and Ivan Uemlianin, "Arweave: A Protocol for Economically Sustainable Information Permanence," 2023. [Online]. Available: https://www.arweave.org/yellow-paper.pdf

[14] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin, "Storj: A Peer-to-Peer Cloud Storage Network," 2014. [Online]. Available: https://storj.io/storj.pdf

[15] Juan Benet, "IPFS - Content Addressed, Versioned, P2P File System," 2014. [Online]. Available: https://ipfs.tech/ipfs.pdf

[16] IPFS Contributors, *When will data be permanently available?*. (2025). GitHub. [Online]. Available: https://github.com/ipfs/ipfs/issues/165

[17] James S. Plank and Lihao Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," 2006.

[18] Srinath Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," 2020. [Online]. Available: https://eprint.iacr.org/2019/550