KONTOR STORAGE PROTOCOL

January 10, 2026

ADAM KRELLENSTEIN
adam@kontor.network

ALEXEY GRIBOV
alexey@kontor.network

OUZIEL SLAMA
ouziel@kontor.network

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 compressed SNARK is constant-size (~12 kB) regardless of the number of challenged symbols; the full proof includes per-file metadata adding ~40 bytes per file. 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. For the cryptographic proof generation and verification algorithms, see the Kontor Proof-of-Retrievability.[3]

Perpetual storage is funded by constant emissions: new KOR tokens are minted at a fixed annual rate (μ_0, relative to total supply) and distributed to storage nodes as rewards. This design is viable because Kontor's cost structure is fundamentally intrinsic. Traditional decentralized storage systems cannot directly observe real-world storage costs or token exchange rates; any mechanism depending on external price feeds introduces centralization risks and manipulation vectors. Kontor solves this by ensuring the dominant cost of storage provision is the opportunity cost of staking KOR, not physical infrastructure. When capital costs dwarf physical storage costs, the system becomes a closed loop: storage providers earn KOR for staking KOR. Node operators compare KOR rewards against KOR staking costs —both intrinsic to the protocol—so no oracle is needed. Market forces naturally balance supply and demand: when profitability falls, nodes exit and rewards concentrate among remaining nodes; when rewards rise, new nodes join and dilute rewards. This equilibrium-finding process maintains adequate replication through natural market dynamics without protocol intervention.

For a high-level overview of the Kontor system as a whole, see the Kontor Whitepaper.[4] For protocol parameters, see the Appendix of the Whitepaper.

2 Protocol

2.1 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. This stake is part of Kontor's unified staking system, where the same KOR can simultaneously back storage commitments, consensus participation, and bridge operations.

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.

2.2 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.

2.3 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.

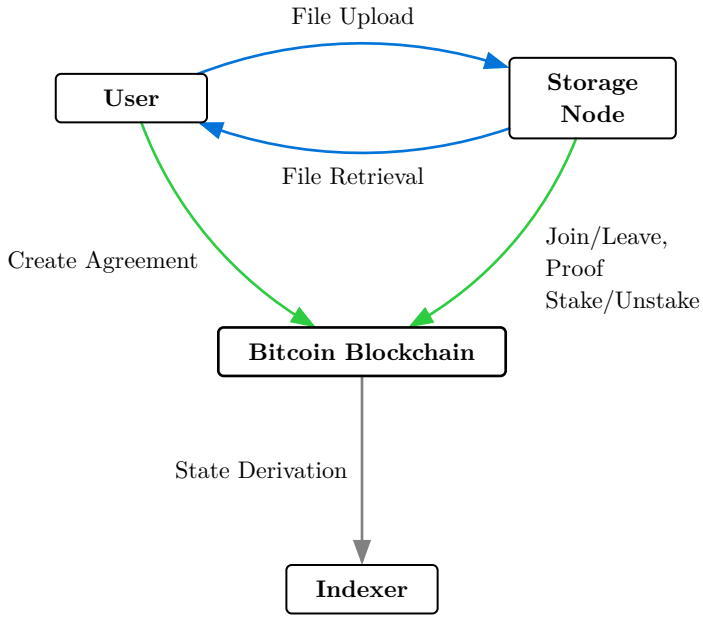

Figure 1: Actor Interaction Model
Blue: off-chain data transfer
Green: transactions
Grey: state derivation

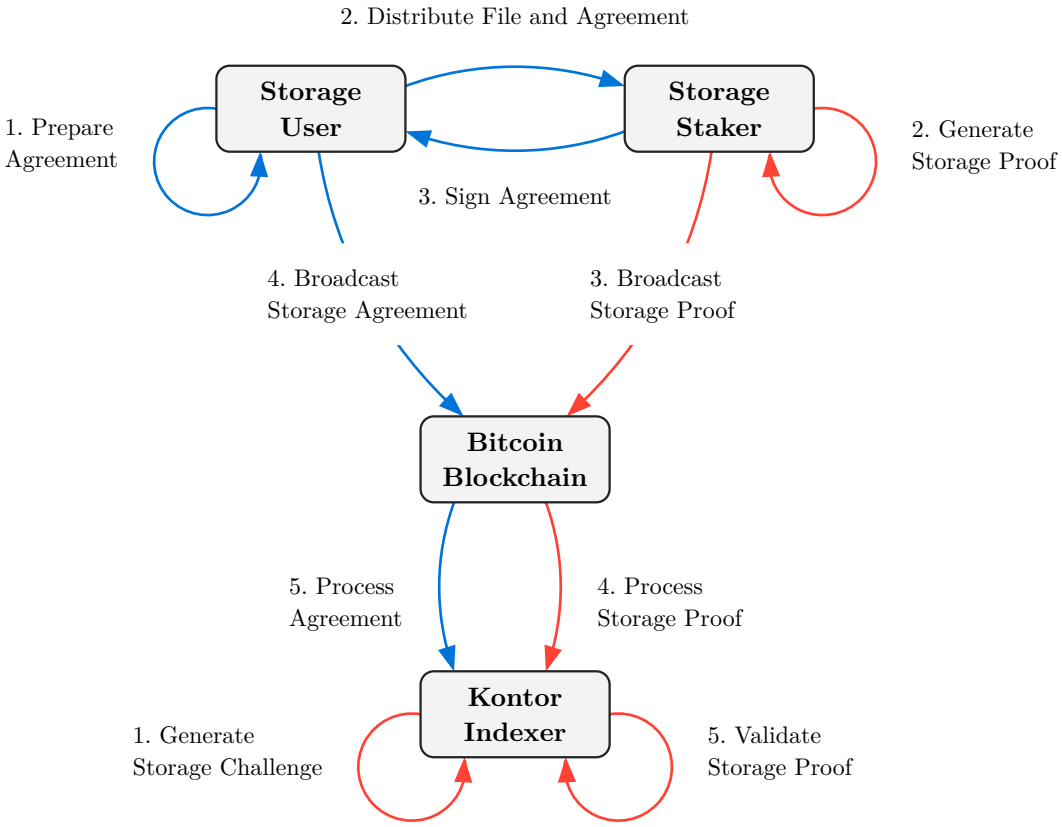

FIGURE 1. File storage flow: agreement creation and challenge-response proof cycle.

2.4 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.

2.4.1 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 2.8.2). 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.

2.4.2 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 `Join-Agreement`):

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

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 2.8.2). This maintains determinism in fee and stake calculations for new files.

2.4.3 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.

2.4.4 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.

2.4.5 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).

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

2.4.6 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:

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

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.

2.4.7 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:

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

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 Bitcoin miner 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.

2.4.8 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:

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

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, see Parameter Selection in the Appendix.

2.4.9 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
4. All files exist at their claimed ledger indices in the proof's ledger root $\rho_{\mathcal{L}}$

Cross-Block Aggregation: The proof includes the ledger root $\rho_{\mathcal{L}}$ used for proof generation. Provers typically use the current ledger root, which contains all files they're being challenged on. The verifier checks that this root is in its set of accepted historical roots (covering at least W_{proof} blocks of file activations). This enables aggregation across challenges from different blocks: even if new files activate during the proof window, proofs generated against an earlier (but still valid) ledger state remain acceptable.

Nodes can aggregate multiple challenges into a single proof transaction to minimize Bitcoin fees. For the cryptographic construction, see the Kontor Proof-of-Retrievability.[3] For economic analysis of proving costs, see Section 3.4.2.

2.4.10 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 Bitcoin miner 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 proof transaction (typically 12-50 kB depending on batch size), 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.

2.4.11 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.

2.5 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

2.5.1 File Agreement Creation Flow

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

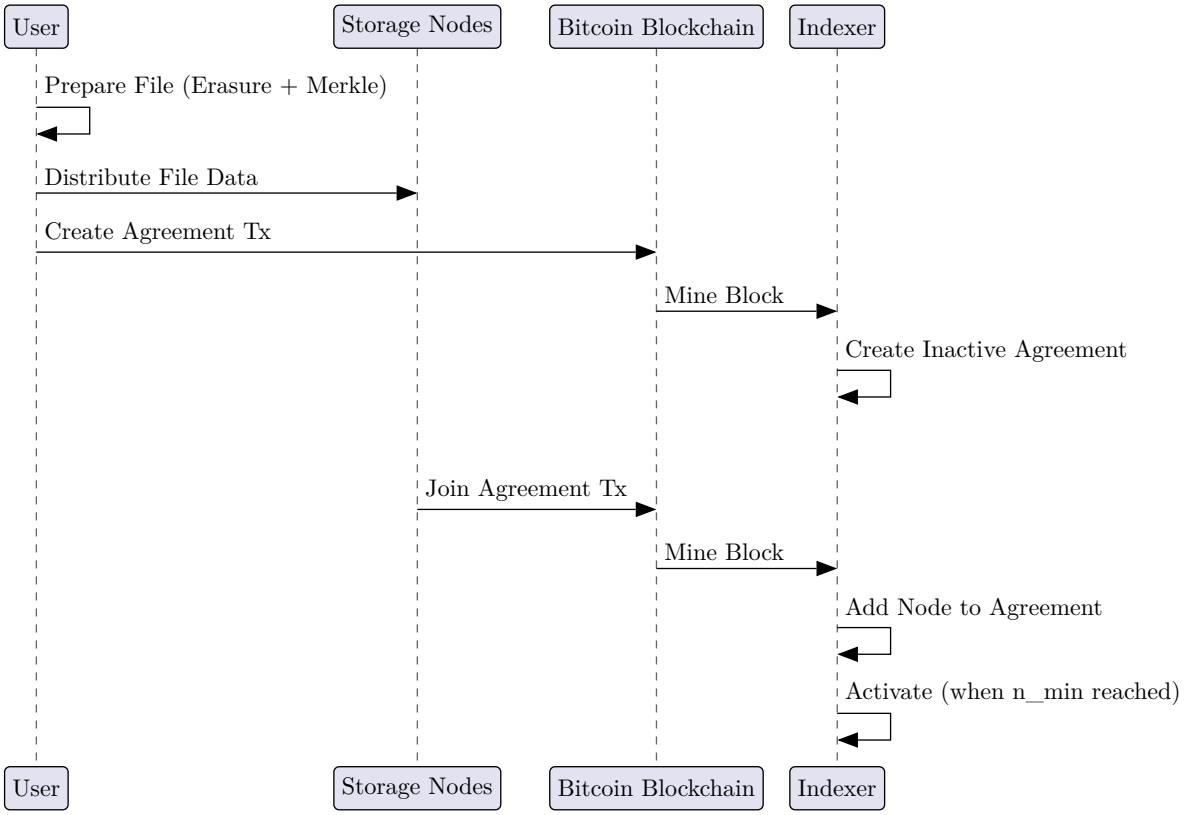

Figure 2: File Agreement Creation Flow. Users prepare files locally with erasure coding and Merkle tree commitment, distribute data to storage nodes off-chain, and broadcast the Create Agreement transaction. The agreement remains inactive until $n_{\min}$ storage nodes join, after which it activates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

2.5.2 Challenge-Response Flow

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

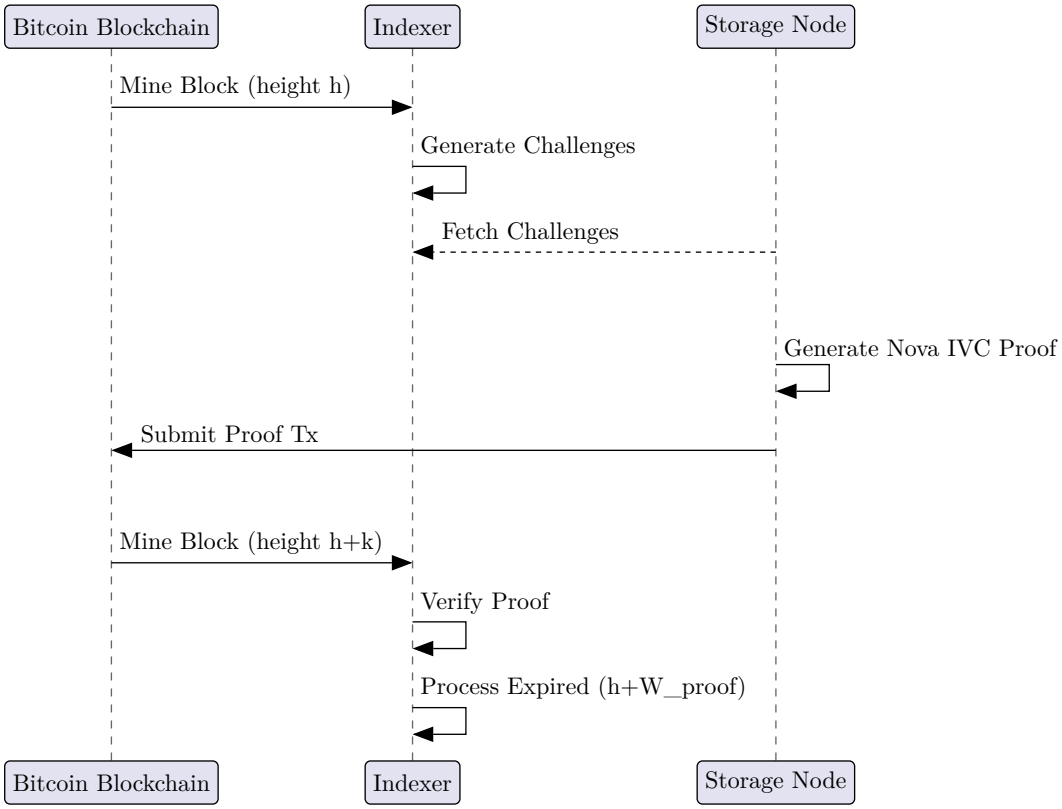

Figure 3: Challenge-Response Flow. Each block, indexers deterministically generate challenges from the block hash. Storage nodes fetch their challenges, generate Nova IVC proofs, and submit them within W_{proof} blocks or face slashing. At block end, the indexer processes expired challenges, slashes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

2.5.3 File Retrieval Flow

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

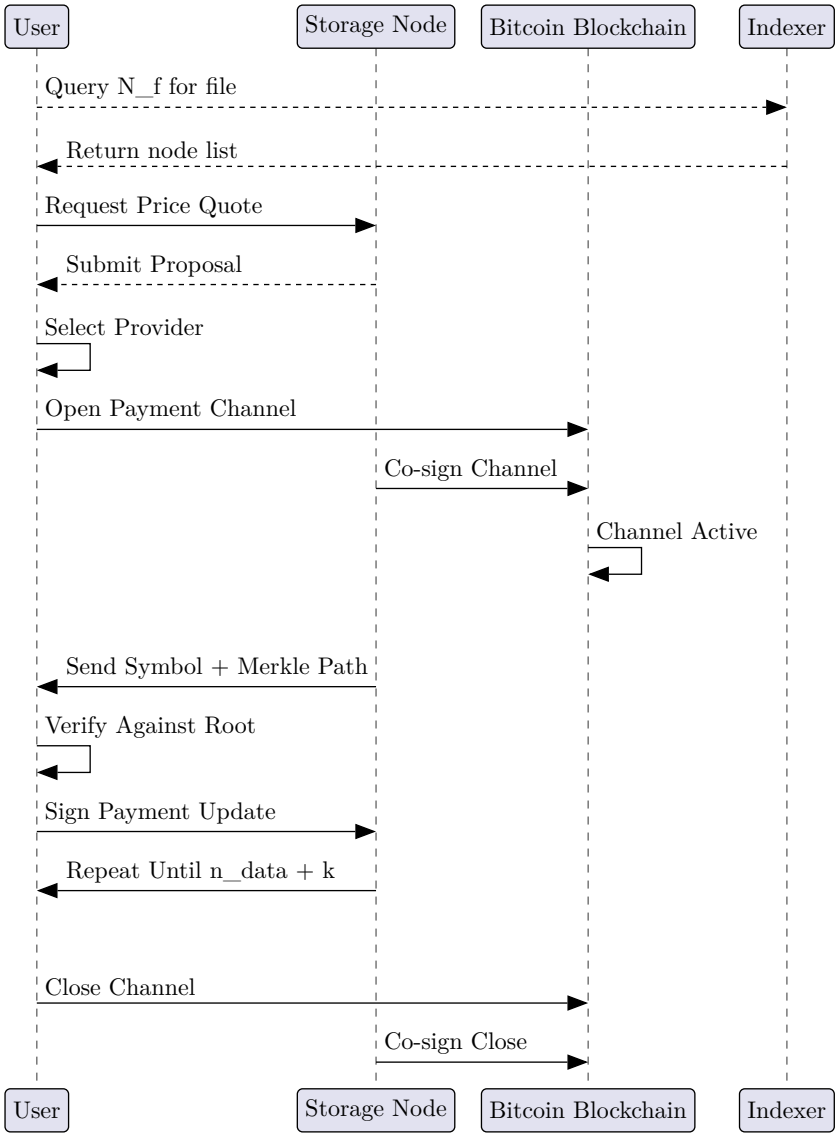

Figure 4: File Retrieval Flow. Users query indexer state for storage nodes ($\mathcal{N}_f$), negotiate pricing off-chain, establish Bitcoin payment channels, and perform atomic symbol-for-payment exchanges. Each symbol is verified against the on-chain Merkle root before payment. The erasure coding structure (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.

2.5.4 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
```

2.5.5 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.

2.5.6 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
        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
```

2.6 Off-Chain Flows

These flows represent voluntary actions taken by users and storage nodes. Transaction creation is voluntary and economically motivated. See Section 3.2 for the economic incentives that drive these behaviors.

2.6.1 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.

2.6.1.1 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 ∈ $[1, n_{\text{codewords}}]$ **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

2.6.1.2 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.

2.6.2 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 payment-channel protocols. 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[5], 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.

2.6.3 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 miner 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).

Edge Cases:

- **Sponsor leaves before acceptance:** If the sponsor leaves the file agreement (voluntarily or via slashing) after posting an offer but before the entrant accepts, the `Join-Agreement` procedure rejects the acceptance (lines 1199–1203 validate that the sponsor is still in the agreement). The entrant loses nothing—they simply cannot accept that particular offer. The entrant should query alternative sponsors or wait for new offers.
- **Sponsor leaves after acceptance:** If the sponsor leaves the file agreement after the entrant has already accepted (sponsorship agreement is active), the sponsorship is voided immediately (see `Leave-Agreement` and slashing procedures). The entrant keeps the file data they already received, retains their position in the file agreement, and the sponsorship commission simply ends early. No commission is paid after the sponsor departs.
- **Multiple offers for same entrant:** An entrant may receive offers from multiple sponsors for the same file. The entrant should compare terms (commission rate, duration, bond requirement) and accept the most favorable. Once one offer is accepted, the entrant joins the agreement; subsequent offers for that entrant/file pair become invalid (the entrant is already a member).

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 balances the sponsor's bandwidth cost against the NPV of commission payments:

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

2.7 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:

2.7.1 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 2.6.1.2) 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
  $\upsilon_f \leftarrow \chi_{\text{fee}} \cdot k_f$
  ▷ Storage fee from economic model

  ▷ 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

2.7.2 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
  $o \leftarrow \{\text{id} : \text{id}_o, \text{file\_id} : \text{file\_id}, \text{sponsor} : \text{signer}, \text{entrant} : \text{entrant\_id}, \text{rate} : \gamma_{\text{rate}}, \text{duration} : \gamma_{\text{duration}}, \text{bond} : \text{bond\_amount}, \text{creation\_block} : \text{block\_height}, \text{expiration\_block} : \text{block\_height} + W_{\text{offer}}\}$
  ▷ 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

2.7.3 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
  $\mathcal{A} \leftarrow$ state.agreements.get(agreement_id)
  if $\mathcal{A} = \perp$ **then**
    return $\perp$ (file agreement not found)
  end

  ▷ Step 2: Check if node already in agreement
  if signer $\in \mathcal{A}$.nodes **then**
    return $\perp$ (already joined)
  end

  ▷ Step 3: Process sponsorship offer if provided
  if offer_id $\neq \perp$ **then**
    ▷ Retrieve and validate offer
    $o \leftarrow$ state.offers.get(offer_id)
    if $o = \perp$ **then**
      return $\perp$ (offer not found)
    end

    ▷ Validate offer is for this entrant and file
    if o.entrant $\neq$ signer $\vee$ o.file_id $\neq \mathcal{A}$.file_id **then**
      return $\perp$ (offer mismatch)
    end

    ▷ Validate offer not expired
    if block_height $\geq o$.expiration_block **then**
      return $\perp$ (offer expired)
    end

    ▷ Validate sponsor still in agreement
    if o.sponsor $\neg \in \mathcal{A}$.nodes **then**
      return $\perp$ (sponsor no longer in agreement)
    end
  end

  ▷ Step 4: Verify stake requirement
  $\mathcal{F}_n$ (node file agreements) $\leftarrow$ state.get_files_for_node(signer)
  $\mathcal{F}_n'$ (projected file agreements) $\leftarrow \mathcal{F}_n \cup \{\mathcal{A}$.file_id$\}$
  k_n (node stake) $\leftarrow$ state.get_stake(signer)
  ▷ Calculate required stake with projected file set
  stake_sum $\leftarrow \sum_{f \in \mathcal{F}_n'}$ state.get_agreement(f).base_stake
  $\lambda_{\text{stake}} \leftarrow 1 + \frac{\lambda_{\text{slash}}}{\ln(2+|\mathcal{F}_n'|)}$
  $k_{\text{required}} \leftarrow$ stake_sum $\cdot \lambda_{\text{stake}}$
  ▷ Total required stake from economic model
  if $k_n < k_{\text{required}}$ **then**
    return $\perp$ (insufficient stake)
  end
  ▷ Predictive check: stake validated using PROJECTED file set $\mathcal{F}_n'$
  ▷ 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 |A.nodes| ≥ n_min ∧ A.active = false then
        A.active ← true
        F.add(A.file_id)
        ▷ File agreement added to F, begins receiving emissions

        ▷ Update global emission weight
        ω_f ← A.emission_weight
        Ω ← state.get_omega()
        Ω ← Ω + ω_f
        state.set_omega(Ω)
        ▷ Increment global Ω when file agreement activates
    end
    return success
end
```

2.7.4 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
  $\mathcal{A} \leftarrow$ state.agreements.get(agreement_id)
  if $\mathcal{A} = \perp \vee$ signer $\neg \in \mathcal{A}$.nodes **then**
    return $\perp$ (invalid request)
  end

  ▷ Step 2: Check minimum nodes constraint
  $\mathcal{N}_f$ (nodes for file agreement) $\leftarrow$ state.get_nodes_for_file($\mathcal{A}$.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
  $k_f \leftarrow \mathcal{A}$.base_stake
  $\varphi_{\text{leave}} \leftarrow k_f \cdot \left(\frac{n_{\min}}{|\mathcal{N}_f|}\right)^2$
  ▷ Leave fee from economic model
  b_n (spendable balance) $\leftarrow$ state.get_balance(signer)
  if $b_n < \varphi_{\text{leave}}$ **then**
    return $\perp$ (insufficient balance for leave fee)
  end

  ▷ Step 4: Burn leave fee from spendable balance
  state.reduce_balance(signer, φ_{leave})
  state.burn(φ_{leave})

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

  ▷ Step 6: Update membership sets
  $\mathcal{N}_f$.remove(signer)
  state.set_nodes_for_file($\mathcal{A}$.file_id, $\mathcal{N}_f$)
  $\mathcal{F}_n$ (file agreements for node) $\leftarrow$ state.get_files_for_node(signer)
  $\mathcal{F}_n$.remove($\mathcal{A}$.file_id)
  state.set_files_for_node(signer, $\mathcal{F}_n$)

  ▷ Step 7: Void sponsorships involving departing node
  for $m \in$ state.sponsorships **do**
    if m.entrant = signer $\vee$ m.sponsor = signer **then**
      state.sponsorships.remove(m)
      ▷ Voided: node was sponsor or entrant
    end
  end
  return success
end

2.7.5 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 one or more challenge identifiers. Proofs may aggregate multiple challenges across multiple files and block heights into a single constant-size SNARK.[3] The verifier reconstructs the expected public inputs deterministically from protocol state (challenges, file ledger roots, indices) and verifies the compressed SNARK.

If the proof is valid, each covered challenge is marked as verified and removed from the active challenge queue. If the proof is invalid or malformed, the proof is rejected and each referenced challenge is immediately marked as failed, triggering slashing during block-end processing.

Algorithm 9: Verify Storage Proof

procedure VERIFY-STORAGE-PROOF(state, signer, proof, block_height)
  ▷ Step 1: Validate challenge IDs and retrieve challenges
  ids ← proof.challenge_ids
  if |ids| = 0 **then**
    return ⊥ (empty proof)
  end
  ▷ Reject duplicate IDs to prevent accidental double-counting
  if |ids| ≠ |Set(ids)| **then**
    return ⊥ (duplicate challenge IDs)
  end
  challenges ← empty list
  for id ∈ ids **do**
    $\mathcal{C}$ ← state.get_challenge(id)
    if $\mathcal{C} = \perp$ **then**
      return ⊥ (challenge not found)
    end
    if signer ≠ $\mathcal{C}$.node_id **then**
      return ⊥ (unauthorized - wrong node)
    end
    if $\mathcal{C} \notin$ state.active_challenges **then**
      return ⊥ (challenge not active)
    end
    challenges.append($\mathcal{C}$)
  end

  ▷ Step 2: Validate ledger root (multi-file only)
  $\rho_{\mathcal{L}}$ ← proof.ledger_root
  d_{agg} ← proof.aggregated_tree_depth
  if $d_{\text{agg}} > 0 \wedge \rho_{\mathcal{L}} \notin$ state.accepted_historical_roots() **then**
    return ⊥ (invalid ledger root)
  end
  ▷ Ledger root validation is skipped for single-file proofs ($d_{\text{agg}} = 0$):
  ▷ the circuit directly enforces computed_root == z0[0], so a mismatched
  ▷ proof.ledger_root causes SNARK verification to fail. Early rejection is optional.

  ▷ Step 3: Determine circuit shape from challenges
  k ← next_power_of_two(|challenges|)
  $d_{\max}$ ← $\max_{\mathcal{C} \in \text{ challenges}} \text{depth}(\mathcal{C})$
  ▷ Use d_{agg} from proof to select parameters

  ▷ Step 4: Reconstruct expected public inputs using proof indices
  $\boldsymbol{I}$ ← proof.ledger_indices
  if $|\boldsymbol{I}| \neq k$ **then**
    return ⊥ (ledger index vector has wrong length)
  end
  ▷ Sort challenges deterministically by (file_id, challenge_id) before building per-file arrays
  sorted ← Sort(challenges, $(c_1, c_2) \Rightarrow c_1$.file_id $< c_2$.file_id $\vee$ (c_1.file_id = c_2.file_id $\wedge$ c_1.id $< c_2$.id))
  $\boldsymbol{D}, \boldsymbol{\Sigma}$ ← arrays from sorted, padded to length k
  $\boldsymbol{z}_0^{\text{expected}} \leftarrow [\rho_{\mathcal{L}}, 0, \boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \mathbf{0}]$

  ▷ Step 5: Generate verification parameters
  pp ← $\mathcal{G}(k, d_{\max}, d_{\text{agg}})$

  ▷ Step 6: Verify compressed SNARK
  $\pi_{\text{compressed}}$ ← proof.compressed_snark
  N ← challenges[0].num_challenges
  if $\exists \mathcal{C} \in$ challenges : $\mathcal{C}$.num_challenges $\neq N$ **then**
    return ⊥ (mismatched iteration count)
  end

valid ← Spartan.Verify$\big(\text{pp}, \pi_{\text{compressed}}, \boldsymbol{z}_0^{\text{expected}}, N\big)$
if ¬ valid **then**
▷ Invalid proof: mark referenced challenges as failed (slashable immediately)
for $\mathcal{C} \in$ challenges **do**
STATE.failed_challenges.add($\mathcal{C}$.id)
STATE.active_challenges.remove($\mathcal{C}$)
end
return $\perp$ (proof verification failed)
end

▷ Step 7: Mark challenges as verified and remove from active list
for $\mathcal{C} \in$ challenges **do**
STATE.verified_challenges.add($\mathcal{C}$.id)
STATE.active_challenges.remove($\mathcal{C}$)
end

▷ Step 8: Release bond(s) for first successful proof(s)
seen_files ← empty set
for $\mathcal{C} \in$ challenges **do**
file_id ← $\mathcal{C}$.file_id
if file_id $\notin$ seen_files **then**
seen_files.add(file_id)
for $m \in$ state.sponsorships **do**
if m.file_id = file_id $\wedge$ m.entrant = signer $\wedge$ 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
end
end
return success
end

2.7.6 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 $\leq 0 \vee$ amount $> b_n$ **then**
return $\perp$ (invalid amount)
end

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

```
    return success
end
```

2.7.7 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 ≤ 0 ∨ amount > k_n then
        return ⊥ (invalid amount)
    end

    ▷ Calculate required stake for node's file agreements
    F_n (node file agreements) ← state.get_files_for_node(signer)
    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

    ▷ Check if withdrawal would violate stake requirement
    k'_n (stake after withdrawal) ← k_n − amount
    if k'_n < k_required then
        return ⊥ (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
```

2.8 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.

2.8.1 Challenge Generation

Kontor indexers deterministically derive a set of challenges from each Bitcoin block using the current block hash as the entropy source. The algorithm proceeds in three steps: (1) derive a batch seed from the block hash using HKDF, (2) select files probabilistically based on a uniform challenge rate $p_f = \frac{C_{\text{target}}}{B}$, and (3) for each selected file, uniformly sample one storage node to be challenged. This ensures each file receives approximately C_{target} challenges per year regardless of network size, while distributing challenge load evenly across nodes.

Algorithm 12: Create Challenge

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

  ▷ Step 2: Set challenge parameters and expiration
  expiration_block $\leftarrow$ block_height $+ W_{\text{proof}}$
  ▷ Must respond within W_{proof} blocks with s symbol proofs

  ▷ Step 3: Package challenge structure
  $\mathcal{C} \leftarrow \{\text{id} : \mathcal{I}, \text{node\_id} : \text{node\_id}, \text{file\_id} : \text{file\_id}, \text{metadata} : \text{metadata}, \text{block\_height} : \text{block\_height}, \text{expiration\_block} : \text{expiration\_block}, \text{num\_challenges} : s, \text{seed} : \sigma_{\text{batch}}\}$
  return $\mathcal{C}$
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
  $H \leftarrow$ block_hash
  info $\leftarrow$ KONTOR-CHAL::v1 $\|$ block_height
  $\sigma_{\text{batch}} \leftarrow \text{HKDF}_{\text{SHA256}(H,\ \text{info})}$
  ▷ Current block hash provides unpredictable entropy

  ▷ Step 2: Calculate challenge probability (constant across all files)
  $p_f \leftarrow \frac{\mathcal{C}_{\text{target}}}{B}$
  ▷ Challenge probability from global parameters

  ▷ Step 3: Probability-based file selection
  $\mathcal{F} \leftarrow$ state.get_files(block_height)
  $\mathcal{F}_{\text{selected}} \leftarrow$ empty list
  for $f \in \mathcal{F}$ **do**
    ▷ Derive file-specific random value
    $u_f \leftarrow \mathcal{H}_{\text{SHA256}(\sigma_{\text{batch}} \,\|\, f.\text{id})} \bmod \frac{2^{32}}{2^{32}}$
    ▷ Uniform $u_f \in [0, 1)$
    if $u_f < p_f$ **then**
      $\mathcal{F}_{\text{selected}}$.append($f$)
    end
  end

  ▷ Step 4: Select one node per challenged file agreement
  ▷ Node $n \in \mathcal{N}_f$ selected with probability $\frac{1}{|\mathcal{N}_f|}$
  $\mathcal{C}_{\text{new}} \leftarrow$ empty list
  for $f \in \mathcal{F}_{\text{selected}}$ **do**
    $\mathcal{N}_f \leftarrow$ state.get_nodes_for_file(f.id)
    if $|\mathcal{N}_f| > 0$ **then**
      $n \leftarrow \text{RandomChoice}(\mathcal{N}_f, \sigma_{\text{batch}})$
      ▷ Uniform random selection: probability $\frac{1}{|\mathcal{N}_f|}$
      $\mathcal{C} \leftarrow$ Create-Challenge(n, f.id, f.metadata, block_height, σ_{batch}, params)
      $\mathcal{C}_{\text{new}}$.append($\mathcal{C}$)
    end
  end
  return $\mathcal{C}_{\text{new}}$
end

2.8.2 Reward Emissions

Every block, the protocol mints new KOR and distributes it to storage nodes proportionally based on their file agreements and sponsorship arrangements. Total emissions use a constant rate:

$$\varepsilon = \text{KOR}_{\text{total}} \cdot \left(\frac{\mu_0}{B}\right) \tag{10}$$

where $\text{KOR}_{\text{total}}$ is the circulating supply at the previous block, μ_0 is the baseline annual inflation rate (e.g., 0.10 for 10%), and B is blocks per year ($\approx 52,560$ with 10-minute blocks). This constant-emission design is viable because capital costs (staking opportunity cost) dominate physical storage costs, creating a self-referential system that requires no external price oracles.

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()
  $\varepsilon \leftarrow \text{KOR}_{\text{total}} \cdot (\frac{\mu_0}{B})$
  ▷ Constant emissions

  ▷ 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

```
            γ_paid ← 0
            γ_earned ← 0

            ▷ Check if node is entrant in active sponsorship for this file
            for m ∈ state.sponsorships do
                if m.file_id = f ∧ m.entrant = n then
                    if block_height ≥ m.t_start ∧ block_height < m.t_start + m.γ_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
```

2.8.3 Challenge Expiration and Slashing

If a storage node either lets an open challenge expire or submits an invalid storage proof for it, 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

```
1:  procedure PROCESS-FAILED-CHALLENGES(state, block_height)
2:      ▷ Identify and process all failed challenges
3:      ▷ Collect expired challenges (not submitted within W_proof blocks)
4:      C_expired ← empty list
5:      for C ∈ state.active_challenges do
6:          if block_height ≥ C.expiration_block then
7:              C_expired.append(C)
8:          end
9:      end
10:
11:     ▷ Move expired challenges to failed queue
12:     for C ∈ C_expired do
13:         STATE.failed_challenges.add(C.id)
14:     end
15:
16:     ▷ Apply penalties to all failed challenges
17:     for challenge_id ∈ state.failed_challenges do
18:         C_failed ← state.challenges.get(challenge_id)
19:         ▷ Find file agreement for challenged file
20:         A_id ← state.get_agreement_for_file(C_failed.file_id)
21:         if A_id ≠ ⊥ then
22:             A ← state.get_agreement(A_id)
23:             node_id ← C_failed.node_id
24:
25:             ▷ Calculate slash penalty: k_f × λ_slash
26:             k_f ← A.base_stake
27:             penalty ← k_f × λ_slash
28:
29:             ▷ Burn and distribute penalty
30:             burn_amount ← penalty × β_slash
31:             distribute_amount ← penalty × (1 − β_slash)
32:             STATE.reduce_stake(node_id, penalty)
33:             STATE.burn(burn_amount)
34:
35:             ▷ Distribute to remaining nodes in file agreement
36:             N_f (other nodes) ← A.nodes \ node_id
37:             if |N_f| > 0 then
38:                 share ← distribute_amount / |N_f|
39:                 for n_i ∈ N_f do
40:                     STATE.add_stake(n_i, share)
41:                 end
42:                 ▷ No remaining nodes: burn entire distribution
43:                 STATE.burn(distribute_amount)
44:             end
45:
46:             ▷ Handle bond transfer if first challenge failure for sponsored join
47:             file_id ← A.file_id
48:             for m ∈ state.sponsorships do
49:                 if m.file_id = file_id ∧ m.entrant = node_id ∧ m.first_proof_complete =
                    false then
50:                     ▷ Entrant failed first challenge: void sponsorship retroactively
51:
52:                     ▷ Transfer bond to sponsor as compensation
53:                     bond ← state.get_bond_escrow((node_id, file_id))
54:                     state.add_balance(m.sponsor, bond)
55:                     state.clear_bond_escrow((node_id, file_id))
56:                     ▷ Sponsor compensated for Bitcoin miner fees and bandwidth costs
57:
```

```
            ▷ 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 + (ln(2 + |F_n^slash|)) / (ln(2 + |F_n|))
        k_required ← stake_sum · λ_stake
        ▷ Total required stake from economic model
        if k_n < k_required then
          ▷ Trigger stake insufficiency handling
          Handle-Stake-Insufficiency(node_id, state)
        end
      end
    end

    ▷ Clear processed failures
    STATE.failed_challenges.clear()
  end
```

2.8.4 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 ($|\mathcal{N}_f| > n_{\min}$). 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 $k_n < k_{\text{req}}$ after slashing
  ▷ Takes precedence over withdrawal operations
  k_n (node stake) ← state.get_stake(node_id)
  $\mathcal{F}_n$ (node file agreements) ← state.get_files_for_node(node_id)
  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
  ▷ Check if insufficiency exists
  if $k_n \geq k_{\text{required}}$ **then**
    return success (no action needed)
  end

  ▷ PASS 1: Graceful Exit from Non-Critical File Agreements
  ▷ Remove from file agreements where $|\mathcal{N}_f| > n_{\min}$
  files_to_remove ← empty list
  for $f \in \mathcal{F}_n$ **do**
    ▷ Check if node can be removed without violating minimum
    $\mathcal{N}_f \leftarrow$ state.get_nodes_for_file(f)
    if $|\mathcal{N}_f| > n_{\min}$ **then**
      files_to_remove.append(f)
    end
  end

  ▷ Shuffle for pseudo-random order
  seed ← $\mathcal{H}_{\text{SHA256(node\_id} \,\|\, \text{block\_height)}}$
  Shuffle(files_to_remove, seed)
  ▷ Prevents node from predicting removal order to game penalties

  ▷ Remove from file agreements until sufficiency restored
  for $f \in$ files_to_remove **do**
    ▷ Get file agreement and calculate penalty
    $\mathcal{A}_{\text{id}} \leftarrow$ state.get_agreement_for_file(f)
    $\mathcal{A} \leftarrow$ state.get_agreement($\mathcal{A}_{\text{id}}$)
    $k_f \leftarrow \mathcal{A}$.base_stake
    penalty ← $k_f \times \lambda_{\text{slash}}$
    ▷ Same penalty as failed challenge: $k_f \times \lambda_{\text{slash}}$

    ▷ Apply penalty for involuntary exit
    burn_amount ← penalty × β_{slash}
    distribute_amount ← penalty × $(1 - \beta_{\text{slash}})$
    STATE.reduce_stake(node_id, penalty)
    $k_n \leftarrow k_n -$ penalty
    ▷ Update local stake tracker
    STATE.burn(burn_amount)

    ▷ Distribute to remaining honest storers
    $\mathcal{N}_f \leftarrow$ state.get_nodes_for_file(f)
    $\mathcal{N}'_f \leftarrow$ empty set
    for $n \in \mathcal{N}_f$ **do**
      if $n \neq$ node_id **then**
        $\mathcal{N}'_f$.add(n)
      end
    end
    if $|\mathcal{N}'_f| > 0$ **then**
      share ← $\frac{\text{distribute\_amount}}{|\mathcal{N}'_f|}$
      for $n_i \in \mathcal{N}'_f$ **do**
        STATE.add_stake(n_i, share)
      end

```
        ▷ No other storers: burn entire distribution
        STATE.burn(distribute_amount)
    end

    ▷ Remove node from file agreement
    $\mathcal{N}_f$.remove(node_id)
    state.set_nodes_for_file($f, \mathcal{N}_f$)
    $\mathcal{F}_n$.remove($f$)
    state.set_files_for_node(node_id, $\mathcal{F}_n$)
    $\mathcal{A}$.nodes.remove(node_id)
    ▷ Node forcibly removed from file agreement

    ▷ Void sponsorships involving removed node for this file
    for $m \in$ state.sponsorships do
        if ($m$.file_id $= f$) $\wedge$ ($m$.entrant = node_id $\vee$ $m$.sponsor = node_id) then
            state.sponsorships.remove($m$)
        end
    end

    ▷ Check if sufficiency restored after this removal
    stake_sum $\leftarrow \sum_{f \in \mathcal{F}_n} \lambda_{\text{slash}}$ state.get_agreement($f$).base_stake
    $\lambda_{\text{stake}} \leftarrow 1 + \frac{}{\ln(2+|\mathcal{F}_n|)}$
    $k_{\text{required}} \leftarrow$ stake_sum $\cdot \lambda_{\text{stake}}$
    ▷ Recompute required stake with updated file agreement set
    if $k_n \geq k_{\text{required}}$ 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 $k_n < k_{\text{required}}$ then
    ▷ Node still insufficient after graceful exits
    ▷ Apply maximum penalty: burn entire remaining stake
    stake_remaining $\leftarrow$ 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
    $\mathcal{F}_n \leftarrow$ state.get_files_for_node(node_id)
    ▷ May include file agreements where $|\mathcal{N}_f| \leq n_{\min}$ (critical agreements)
    for $f \in \mathcal{F}_n$ do
        ▷ Update node membership for this file agreement
        $\mathcal{N}_f \leftarrow$ state.get_nodes_for_file($f$)
        $\mathcal{N}_f$.remove(node_id)
        state.set_nodes_for_file($f, \mathcal{N}_f$)

        ▷ Remove from file agreement structure
        $\mathcal{A}_{\text{id}} \leftarrow$ state.get_agreement_for_file($f$)
        $\mathcal{A} \leftarrow$ state.get_agreement($\mathcal{A}_{\text{id}}$)
        $\mathcal{A}$.nodes.remove(node_id)
        ▷ File agreement may now be under-replicated if $|\mathcal{N}_f| < n_{\min}$, but remains in $\mathcal{F}$ 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 Economic Analysis

This section analyzes the economic incentives and decision-making framework for storage nodes. The analysis provides formal models for node profitability, optimal behavior, and detection guarantees.

3.1 Storage Node Economics

3.1.1 Modeling Parameters

These parameters are used for economic analysis and node decision-making, and may vary based on market conditions or individual node strategies.

- p_{fail}: Probability of an honest storage node failing a challenge (due to operational issues).
- δ: Opportunity cost rate (discount rate) in USD terms per block (per-block fractional interest rate). Annualized rate: $r_{\text{annual}} \approx (1+\delta)^B - 1$. Conversely $\delta \approx (1+r_{\text{annual}})^{\frac{1}{B}} - 1$. For example, a 20% annual rate corresponds to $\delta = (1.20)^{\frac{1}{52560}} - 1 \approx 0.0000034$ per block.
- h: Number of future blocks considered by nodes for profit evaluation (NPV calculation). Typical values range from 2,016 blocks (two weeks) to 26,280 blocks (six months), depending on the node operator's planning horizon and risk tolerance.
- π_{min}: Minimum net expected profit per block in USD that a node requires to stay in an agreement.
- $\xi_{\text{KOR/USD}}$: Exchange rate from KOR to USD.
- $\xi_{\text{BTC/USD}}$: Exchange rate from BTC to USD.
- $\varphi^{\text{rate}}_{\text{BTC}}(t)$: Bitcoin transaction fee rate in BTC per vByte at block t. This is a market-determined value that fluctuates based on Bitcoin network congestion.

3.1.2 Economic Functions

- **Net Present Value calculation for time-varying cashflows:**

$$\text{NPV}(\{c_i\}, \delta, h) \stackrel{\text{def}}{\equiv} \sum_{i=1}^{h} \left(\frac{c_i}{(1+\delta)^i} \right) \tag{11}$$

  where $\{c_i\}$ represents the sequence of expected cashflows in USD for blocks $i = 1, 2, ..., h$.

3.1.3 Cost Functions

- **Expected Costs for node n for file f at time t (all in USD):**

This model focuses on the primary on-chain economic costs. It does not formally include secondary operational costs such as hardware amortization, bandwidth for data repair and propagation, or manual labor, which must also be considered by node operators.

- Expected proving cost: With aggregated proofs, a node pays once per block if challenged on any files: Let $P_{\text{any}}(n,t)$ be the probability that node n is challenged on at least one file. Let $c^{\text{BTC}}_{\text{proof}}(t) \stackrel{\text{def}}{=} s_{\text{proof}} \cdot \varphi^{\text{rate}}_{\text{BTC}}(t)$ be the cost of submitting an aggregated proof in BTC.

$$E\big[c^{\text{USD}}_{\text{prove}}(n,t)\big] \stackrel{\text{def}}{=} P_{\text{any}}(n,t) \cdot c^{\text{BTC}}_{\text{proof}}(t) \cdot \xi_{\text{BTC/USD}} \tag{12}$$

  where $P_{\text{any}}(n,t) \stackrel{\text{def}}{=} 1 - \Pi_{f \in \mathcal{F}_n}\Big(1 - \Big(\frac{p_f}{|\mathcal{N}_f|}\Big)\Big)$. Note that this assumes independent challenge selection across files, which is pseudo-random. Assuming equal replication $|\mathcal{N}_f| = n$ for all f, this simplifies to $1 - \Big(1 - \frac{p_f}{n}\Big)^{|\mathcal{F}_n|}$.

- Expected slashing cost:

$$E\big[c^{\text{USD}}_{\text{slash}}(f,t)\big] \stackrel{\text{def}}{=} P_{\text{chal}}\big(|\mathcal{N}_f|,t\big) \cdot p_{\text{fail}} \cdot k_f \cdot \lambda_{\text{slash}} \cdot \xi_{\text{KOR/USD}} \tag{13}$$

  Note: Slashing occurs when a node is challenged but chooses not to respond with a proof, or when the proof is invalid, or when the proof is not included on-chain within the W_{proof} block window.

- Physical storage cost:

$$c^{\text{USD}}_{\text{storage}}(f,t) \stackrel{\text{def}}{=} s_f^{\text{bytes}} \cdot c^{\text{USD}}_{\text{byte-block}} \tag{14}$$

  where $c^{\text{USD}}_{\text{byte-block}}$ is the marginal cost of storing one byte for one block. While typically small, this cost scales linearly with file size, in contrast to the logarithmic scaling of rewards.

3.1.4 Profit Functions

- **Net Expected Profit for node n at time t (USD):** The total profit for a node is the sum of profits from all files minus the aggregated proving cost and the opportunity cost on the total required stake:

$$\begin{aligned} E[\pi(n,t)] \stackrel{\text{def}}{=} \sum_{f \in \mathcal{F}_n} \Big[\big(E\big[r_{\text{storage}}(n,f,t)\big] + E[r_{\text{slash}}(n,f,t)]\big) \cdot \xi_{\text{KOR/USD}} \\ - E\big[c^{\text{USD}}_{\text{slash}}(f,t)\big] - c^{\text{USD}}_{\text{storage}}(f,t)\Big] - E\big[c^{\text{USD}}_{\text{prove}}(n,t)\big] - k_{\text{req}}(n) \cdot \xi_{\text{KOR/USD}} \cdot \delta \end{aligned} \tag{15}$$

  The per-file profit (excluding the shared proving cost and total stake opportunity cost) is:

$$\begin{aligned} E[\pi(n,f,t)] \stackrel{\text{def}}{=} \big(E\big[r_{\text{storage}}(n,f,t)\big] + E[r_{\text{slash}}(n,f,t)]\big) \cdot \xi_{\text{KOR/USD}} \\ - E\big[c^{\text{USD}}_{\text{slash}}(f,t)\big] - c^{\text{USD}}_{\text{storage}}(f,t) \end{aligned} \tag{16}$$

  This represents the direct profit from a single file agreement before accounting for costs shared across the node's entire portfolio (aggregated proving and total stake opportunity cost).

3.2 Node Decision Framework

Storage nodes make decisions to maximize their expected utility (measured in USD profit).

- **Decision to Join File f:** A node n not currently storing f considers joining if the marginal impact on its total profit is positive. Let $E[\pi(n,t \mid F_n)]$ denote the node's total expected profit given its current file set $\mathcal{F}_n$. The node joins if:

$$\text{NPV}(\{E[\pi(n, t+i \mid F_n \cup \{f\})] - E[\pi(n, t+i \mid F_n)]\}, \delta, h) > 0 \tag{17}$$

This marginal profit calculation captures:

- The additional storage rewards from file f
- The change in total stake opportunity cost due to increased $k_{\text{req}}(n)$ and potential change in $\lambda_{\text{stake}}(n)$
- The change in expected proving costs due to increased $P_{\text{any}}(n, t)$
- The physical storage cost for file f

For this decision, the node must forecast future values of:

- Exchange rates ($\xi_{\text{KOR/USD}}$, $\xi_{\text{BTC/USD}}$)
- Network replication levels ($|\mathcal{N}_f|$)
- Bitcoin fee rates ($\varphi^{\text{rate}}_{\text{BTC}}$)
- Network size and emission parameters

The analysis assumes nodes make rational forecasts based on current trends and market conditions. The participant count is assumed to be $|\mathcal{N}_f| + 1$ (including the joining node).

- **Decision to Sponsor Node for File f:** An existing node n in $\mathcal{N}_f$ decides whether to sponsor an Entrant by posting a sponsorship offer on-chain. The sponsor specifies commission rate γ_{rate}, duration γ_{duration}, and required bond β_{bond}. The sponsor's costs are: (1) Bitcoin transaction fees to post the offer, and (2) bandwidth to transfer file data after acceptance. The benefit is the NPV of the commission stream. The sponsor is protected by the bond: if the entrant fails the first challenge, the bond compensates the sponsor's costs and the sponsorship voids.

  Assuming the entrant receives valid data and proves successfully (the expected case), the sponsor's payoff is:

$$\text{NPV}\left(\left\{\gamma_{\text{rate}} \cdot \left(\varepsilon_f \frac{t+i}{|\mathcal{N}_f|+1}\right)\right\}, \delta, \gamma_{\text{duration}}\right) - c^{\text{offer}}_{\text{BTC}} \cdot \xi_{\text{BTC/USD}} - c^{\text{USD}}_{\text{transfer}} \tag{18}$$

  If the entrant fails the first challenge (invalid data or operational failure), the sponsor receives the bond: $\beta_{\text{bond}} \cdot \xi_{\text{KOR/USD}} - c^{\text{offer}}_{\text{BTC}} \cdot \xi_{\text{BTC/USD}} - c^{\text{USD}}_{\text{transfer}}$. The sponsor sets β_{bond} to cover costs: $\beta_{\text{bond}} \cdot \xi_{\text{KOR/USD}} \geq c^{\text{offer}}_{\text{BTC}} \cdot \xi_{\text{BTC/USD}} + c^{\text{USD}}_{\text{transfer}}$.

  This creates a competitive market for sponsorship. A cartel of existing nodes attempting to block entry (gatekeep) will be broken by the incentive for any single member to defect and capture the commission. The bond-escrow mechanism prevents griefing while maintaining trustless operation.

- **Decision to Leave File f:** A node n currently storing f evaluates whether its total profit would improve by leaving. The node compares:
  - The immediate cost: leave fee $\varphi_{\text{leave}}(f, t) \cdot \xi_{\text{KOR/USD}}$
  - The benefit: NPV of the improvement in total profit from leaving

  The node leaves if:

$$\text{NPV}(\{E[\pi(n, t+i \mid F_n \setminus \{f\})] - E[\pi(n, t+i \mid F_n)]\}, \delta, h) > -\varphi_{\text{leave}}(f, t) \cdot \xi_{\text{KOR/USD}} \tag{19}$$

  Note that the left side is typically negative (leaving reduces profit), so this condition checks if the profit reduction is less than the leave fee. The node may also be subject to a minimum profitability constraint $\pi_{\min}$, leaving if its total profit falls below this threshold and the above condition is met. Equivalently, with an explicit per-block profit floor $\pi_{\min}$ (USD/block), a node leaves when $E[\pi(n, t)] < \pi_{\min}$ (or $E[\pi(n, t)] < 0$ if $\pi_{\min} = 0$) and the NPV condition above holds.

  When $|\mathcal{N}_f| \leq n_{\min}$, leaving is not permitted regardless of profitability.

- **Decision to Respond to Challenge:** When node n is challenged for file f, it chooses to generate a proof if the cost of proving is less than the total loss from being slashed.

  Analysis: The true decision must account for the loss of all future revenue streams. Being slashed removes the node from the agreement permanently, forfeiting future rewards. Using a finite analysis horizon h and assuming parameters are roughly constant over that horizon, a simplified estimate of this loss is given by 20:

$$\begin{aligned} \text{NPV}_{\text{future}} &\approx \sum_{i=1}^{h} \left(\frac{\varepsilon_f(t+i) \cdot \xi_{\text{KOR/USD}}}{|\mathcal{N}_f| \cdot (1+\delta)^i} \right) \\ &\approx \left(\frac{\varepsilon_f(t) \cdot \xi_{\text{KOR/USD}}}{|} \mathcal{N}_f| \right) \cdot \frac{1-(1+\delta)^{-h}}{\delta} \end{aligned} \tag{20}$$

  In reality, $|\mathcal{N}_f|$, $\varepsilon_f(t)$, and $\xi_{\text{KOR/USD}}$ will vary over time. The node must forecast these changes when making its decision.

  All terms in the inequalities below are expressed in USD (NPV); KOR flows are converted at $\xi_{\text{KOR/USD}}$ and BTC costs at $\xi_{\text{BTC/USD}}$.

  The decision criterion is then given by 21:

$$c_{\text{proof}}^{\text{BTC}}(t) \cdot \xi_{\text{BTC/USD}} < k_f \cdot \lambda_{\text{slash}} \cdot \xi_{\text{KOR/USD}} + \text{NPV}_{\text{future}} \tag{21}$$

3.3 Macroeconomic Stability

The protocol's long-term stability relies on intrinsic market dynamics. With constant emissions, stability emerges from the natural equilibrium-seeking behavior of rational nodes responding to market signals.

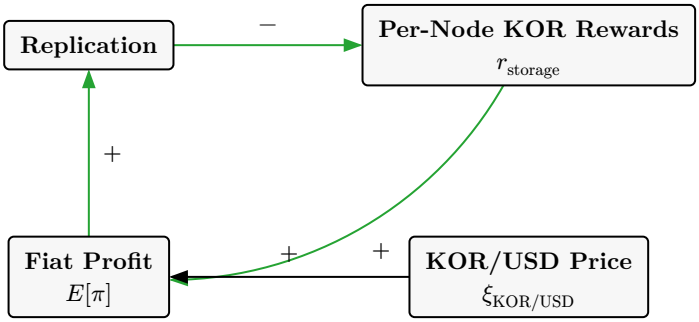

Figure 5: Intrinsic market equilibrium. The **green loop** is stabilizing: as replication falls, per-node rewards increase hyperbolically (fixed emissions split among fewer nodes), improving profitability and attracting new nodes. This equilibrium-seeking process operates without protocol intervention.

The **stabilizing negative feedback loop** operates through market forces:

1. When replication falls (nodes exit), the fixed emissions are split among fewer remaining nodes.
2. Per-node rewards increase, improving profitability for remaining nodes.
3. Higher profitability attracts new nodes, restoring replication toward equilibrium.
4. Conversely, when replication is high, per-node rewards are diluted, naturally discouraging over-replication.

External KOR price movements affect fiat profitability but do not trigger protocol-level inflation responses that could cause death spirals. If KOR price falls:

- Fiat profitability decreases, causing marginal nodes to exit
- Replication may decline as acceptable graceful degradation

- The protocol continues to function as long as $n_{\min}$ replication is maintained
- No inflation response amplifies the price decline

This design accepts variable replication in exchange for monetary stability.

3.4 Capital Cost Dominance

The protocol's security model fundamentally relies on capital costs (staking and proving) dominating physical storage costs. This economic asymmetry prevents various attacks where nodes might attempt to collect rewards without actually storing data.

3.4.1 Stake Requirements

The per-node base stake for a file is determined by the emission weight and network scale. This formula ensures that:

- Stake requirements scale proportionally with the file's share of network emissions
- The logarithmic scaling in network size provides reasonable growth while preventing excessive barriers
- Earlier files (lower rank) require higher stakes, reflecting their greater emission value

With typical parameters, the capital cost of staking significantly exceeds storage costs. For example, at $|\mathcal{F}(t)| = 10^9$ files, the annual opportunity cost of staking (~$0.05) exceeds the physical storage cost (~$0.01 for 100MB) by 5x, ensuring that honest storage remains more profitable than attempting to fake it.

3.4.2 Proving Costs and Aggregation

The protocol's proof aggregation mechanism creates economies of scale that benefit legitimate operators while maintaining security. Nodes have a W_{proof} window (approximately 2 weeks) to submit proofs after being challenged, allowing them to aggregate multiple challenges into a single Bitcoin transaction.

With a per-file challenge probability of $p_f = \frac{C_{\text{target}}}{B} = \frac{12}{52560} \approx 0.000228$ per block, a node storing $|\mathcal{F}_n|$ files expects approximately $\left(\sum_{f \in \mathcal{F}_n} \frac{p_f}{|\mathcal{N}_f|}\right) \times 2016$ challenges over the 2-week window. Under equal replication $|\mathcal{N}_f| = r$ for all f, this becomes $|\mathcal{F}_n| \times \left(\frac{p_f}{r}\right) \times 2016 \approx |\mathcal{F}_n| \times \left(\frac{0.46}{r}\right)$. By aggregating these into a single proof transaction, nodes pay Bitcoin fees only once regardless of the number of challenges.

This aggregation opportunity:

- Makes proving costs negligible even for small operators
- Allows nodes to optimize fee payment timing based on Bitcoin network conditions
- Creates no disadvantage for honest nodes while maintaining security guarantees
- Reduces the risk of network congestion and high BTC proving fees

3.5 Failure Detection

The protocol's challenge mechanism provides strong probabilistic guarantees for detecting data loss, regardless of file size. By sampling a fixed number of sectors ($s_{\text{chal}} = 100$) from each challenged file, the protocol ensures uniform security properties across all stored data.

3.5.1 Detection Probability Analysis

The key insight is that detection probability depends only on the **fraction** of missing data, not the absolute file size. For a file missing fraction ν of its sectors:

- Exact (hypergeometric, without replacement):

$$P(\text{detection} \mid \text{challenged}) = 1 - \Pi_{i=0}^{s_{\text{chal}}-1} \left(\frac{(1-\nu) \cdot s_f - i}{s_f - i} \right) \tag{22}$$

- Binomial approximation (valid when s_f is large and $s_{\text{chal}} \ll s_f$):

$$P(\text{detection} \mid \text{challenged}) \approx 1 - (1 - \nu)^{s_{\text{chal}}} \tag{23}$$

For example, if a node deletes half of a file's data ($\nu = 0.5$) and the protocol challenges just two sectors ($s_{\text{chal}} = 2$), the approximate probability of detection is $1 - (1 - 0.5)^2 = 75\%$.

Each file expects $C_{\text{target}} = 12$ challenges per year, regardless of network size. This constant challenge rate ensures predictable security properties as the network scales.

Complete Loss Detection: For a completely missing file (100% data loss), the annual detection probability follows a Poisson process:

$$P_{\text{annual detection}} = 1 - e^{-C_{\text{target}}} = 1 - e^{-12} \approx 99.9994\% \tag{24}$$

Partial Loss Detection: For partial data loss, we combine the per-challenge detection probability with the expected number of annual challenges. With $\nu = 0.1$ (10% data loss) and $s_{\text{chal}} = 100$, we first calculate the per-challenge detection probability:

$$P(\text{detection} \mid \text{challenged}) = 1 - (1 - 0.1)^{100} = 1 - 0.9^{100} \approx 0.99997 \tag{25}$$

The annual detection probability is:

$$P_{\text{annual detection}} = 1 - e^{-C_{\text{target}} \cdot P(\text{detection} \mid \text{challenged})} \approx 1 - e^{-12 \times 0.99997} \approx 99.9994\% \tag{26}$$

With a high number of challenged sectors, the detection probability for even 10% data loss is so high that the annual detection rate is difficult to distinguish from that of complete data loss. The protocol assumes files are erasure-coded such that all data can be recovered from this level of degradation.

Beyond this threshold, data loss is detected with near-certain probability. With $s_{\text{chal}} = 100$, storing only 90% of the data (at the erasure coding threshold) results in 99.997% detection probability per challenge, implying an expected time before detection of approximately 10.0 months:

$$E[\text{blocks to detection}] = \frac{1}{P_{\text{chal}}(|\mathcal{N}_f|, t) \cdot [1 - (1 - \nu)^{s_{\text{chal}}}]} \tag{27}$$

For typical values with $|\mathcal{N}_f| = 10$, $p_f = 0.000228$, and $\nu = 0.1$:

$$E[\text{blocks to detection}] \approx \frac{1}{0.0000228 \cdot 0.99997} \approx 43,860 \text{ blocks} \approx \left(43, \frac{860}{52}, 560\right) \times 12 \approx 10.0 \text{ months} \tag{28}$$

4 Security Analysis

4.1 Protocol Security

4.1.1 System Invariants

The protocol maintains the following invariants that must hold at all times:

1. **Conservation of KOR:** The total KOR in the system changes according to emissions and burns. At any block t:

$$\sum_{n \in \mathcal{N}} (b_n + k_n) + \text{KOR}_{\text{other}}(t) = \text{KOR}_{\text{total}}(t) \tag{29}$$

where:

- $\sum_{n \in \mathcal{N}} (b_n + k_n)$ represents all KOR held by storage nodes (balances + stakes)
- $\text{KOR}_{\text{other}}(t)$ represents KOR held by users, protocol treasury, exchanges, and other entities outside the storage node ecosystem
- $\text{KOR}_{\text{total}}(t)$ represents the net circulating supply at block t, which evolves according to:

$$\text{KOR}_{\text{total}}(t) = \text{KOR}_{\text{initial}} + \sum_{i=1}^{t} (\varepsilon(i) - \Phi_{\text{burned}}(i)) \tag{30}$$

Let:

- $F_{\text{created}}(t)$ be the set of files created in block t
- $S(t)$ be the set of slashing events (failed challenges) in block t, where each event $s \in S(t)$ involves a node storing file f_s
- $L(t)$ be the set of leave events in block t, where each event $l \in L(t)$ involves a node leaving file f_l

The burns in block t are:

$$\Phi_{\text{burned}}(t) = \sum_{f \in F_{\text{created}}(t)} \upsilon_f + \sum_{s \in S(t)} \beta_{\text{slash}} \cdot k_{f_s} \cdot \lambda_{\text{slash}} + \sum_{l \in L(t)} \varphi_{\text{leave}}(f_l, t) \tag{31}$$

The cumulative burns through block t are:

$$\Phi_{\text{burned}}^{\text{total}}(t) = \sum_{i=0}^{t} \Phi_{\text{burned}}(i) \tag{32}$$

where $\Phi_{\text{burned}}(0) = 0$ at genesis.

2. **Non-negative Balances:** For all nodes $n \in N$:

$$b_n \geq 0 \tag{33}$$

If any operation would result in $b_n < 0$, the operation fails.

3. **Bidirectional Consistency:** For all nodes $n \in N$ and files $f \in F$:

$$f \in \mathcal{F}_n \Leftrightarrow n \in \mathcal{N}_f \tag{34}$$

4. **Positive Parameters:** All protocol parameters must be positive where specified:

$$\mu_0 > 0 \tag{35}$$

$$c_{\text{stake}} > 0 \tag{36}$$

$$0 \leq \delta \tag{37}$$

$$0 \leq p_{\text{fail}} < 1 \tag{38}$$

$$n_{\text{min}} \geq 1 \tag{39}$$

$$\lambda_{\text{slash}} > 0 \tag{40}$$

5. **Stake Sufficiency:** All nodes must maintain a total stake greater than or equal to their total required stake.

$$\forall n \in \mathcal{N} : k_n \geq k_{\text{req}}(n) \tag{41}$$

The protocol programmatically prevents withdrawals that would violate this invariant. If a node's stake becomes insufficient (e.g., after the node is slashed), the automated removal algorithm is triggered to restore sufficiency.

6. **Staked KOR Bound:** The total staked KOR across all nodes is always less than or equal to the total KOR supply:

$$\sum_{n \in \mathcal{N}} k_n \leq \text{KOR}_{\text{total}}(t) \tag{42}$$

This invariant ensures that stake requirements cannot exceed the available KOR supply.

4.2 Economic Security

This section analyzes potential attack vectors against the protocol's economic mechanisms, their profitability, and mitigation strategies.

4.2.1 Attacks on Storage Provision

These attacks involve storage nodes failing to meet their obligations, either to increase profit or to save costs.

4.2.1.1 Selective Storage Attack

A malicious node attempts to maximize profit by storing only a fraction of the file data, gambling that the randomly selected challenge sectors will be among those it has retained.

Risk-Reward Analysis: The attacker must weigh the marginal storage cost savings against the risk of detection and total loss. For a node storing fraction $(1-\nu)$ of file f:

Savings per block: $\nu \cdot c^{\text{USD}}_{\text{storage}}(f,t)$ (only the marginal storage cost)

Risk per block: $P_{\text{chal}}\big(|\mathcal{N}_f|,t\big) \cdot \left[1-(1-\nu)^{s_{\text{chal}}}\right] \cdot \left[k_f \cdot \lambda_{\text{slash}} \cdot \xi_{\text{KOR/USD}} + \text{NPV}_{\text{future}}\right]$

where:

- $P_{\text{chal}}\big(|\mathcal{N}_f|,t\big) = \frac{p_f}{|\mathcal{N}_f|}$ is the probability of being challenged
- $\left[1-(1-\nu)^{s_{\text{chal}}}\right]$ is the probability of detection if challenged
- The loss includes both the slashed stake and all future rewards (NPV), as defined in Eq. 20.

Files are assumed to be erasure-coded to tolerate up to 10% data loss. However, nodes that allow data to degrade to this threshold face near-certain detection. With $s_{\text{chal}} = 100$, storing only 90% of the data results in 99.997% detection probability per challenge. The expected time to detection is approximately 10.0 months for typical values.

The attacker loses all future rewards upon detection, making the NPV of the attack negative even with high discount rates.

4.2.1.2 Collusion Attack

Multiple nodes (k of n total) storing a file coordinate to have one member fail a challenge, aiming to profit from the redistributed stake.

Analysis: Let $n = |\mathcal{N}_f|$ and let the colluding subset have size k. One colluder n_s intentionally fails a challenge.

- Slashed amount: $S = k_f \cdot \lambda_{\text{slash}}$ is deducted from n_s.
- Burn: $S_{\text{burned}} = \beta_{\text{slash}} \cdot S$ is burned.
- Redistribution: $S_{\text{redist}} = (1-\beta_{\text{slash}}) \cdot S$ is split equally among the $(n-1)$ remaining nodes.

The colluding group's net change is: $\Delta\text{KOR}_{\text{colluders}} = -S + \left(\frac{k-1}{n-1}\right) \cdot S_{\text{redist}} = -S + \left(\frac{k-1}{n-1}\right) \cdot (1-\beta_{\text{slash}}) \cdot S$

Profitability would require $\left(\frac{k-1}{n-1}\right) \cdot (1-\beta_{\text{slash}}) > 1$, which is impossible since each factor is ≤ 1. Hence, this collusion is strictly unprofitable.

4.2.1.3 Disk-Sharing Attack

An attacker creates multiple Sybil node identities but stores only a single physical copy of the data, aiming to collect multiple rewards for a single storage cost while undermining the protocol's data replication guarantees.

Analysis: Consider an attacker creating n_{sybil} identities to store the same file f. The attacker's costs are:

- Physical storage: $c^{\text{USD}}_{\text{storage}}(f,t)$ (only one copy needed)

- Stake opportunity cost: $n_{\text{sybil}} \cdot k_f \cdot \xi_{\text{KOR/USD}} \cdot \delta$ (full stake per identity)
- Expected proving costs: $n_{\text{sybil}} \cdot P_{\text{chal}}\big(|\mathcal{N}_f|, t\big) \cdot c_{\text{proof}}^{\text{BTC}}(t) \cdot \xi_{\text{BTC/USD}}$ (each identity may be challenged)

The attacker's revenue is $n_{\text{sybil}} \cdot r_{\text{storage}}(n, f, t) \cdot \xi_{\text{KOR/USD}}$. At the margin (adding one more identity), the change is approximately:

$$\Delta\pi \approx \frac{\varepsilon_f(t) \cdot \xi_{\text{KOR/USD}} \cdot |\mathcal{N}_f|}{\big(|\mathcal{N}_f| + n_{\text{sybil}}\big)^2} - k_f \cdot \delta \cdot \xi_{\text{KOR/USD}} \tag{43}$$

As $|\mathcal{N}_f|$ or n_{sybil} grow, the first term shrinks as $\frac{1}{(|\mathcal{N}_f| + n_{\text{sybil}})^2}$ while the second term is constant, so beyond a small scale the attack is unprofitable even when saving on c_{storage}.

Mitigation: Capital costs dominate storage costs, making the attack unprofitable.

4.2.2 Attacks on Market Mechanics

These attacks exploit the protocol's economic rules to manipulate outcomes or gain an unfair advantage.

4.2.2.1 Sybil Attack (Risk Compartmentalization)

An attacker creates multiple node identities to limit their downside risk from correlated failures.

Analysis: The protocol defends against this by making portfolio splitting more capital-intensive. The dynamic stake factor, $\lambda_{\text{stake}} = 1 + \frac{\lambda_{\text{slash}}}{\ln(2 + |\mathcal{F}_n|)}$, imposes a capital premium on nodes with fewer files.

Example with $\lambda_{\text{slash}} = 30.0$ and an operator with a 100,000-file portfolio:

Node Profile	**Files $\lvert\mathcal{F}_n\rvert$**	**Stake Factor (λ_{stake})**	**Required Stake (k_{req})**
Sybil Node	1	28.30x	28.30 KOR
Small Node	10	13.56x	135.6 KOR
Large Consolidated Node	100,000	3.60x	360,000 KOR

To run the portfolio as 100,000 individual Sybil nodes would require 2,830,000 KOR vs 360,000 KOR for a single entity—a ~7.9x capital increase, rendering large-scale compartmentalization attacks economically irrational.

4.2.2.2 Sybil-Based Reward Amplification

An attacker introduces a Sybil identity to collect an additional share of a file's rewards.

Analysis: Let $|\mathcal{N}_f|$ be nodes before the Sybil joins, and $\lambda_S = 1 + \frac{\lambda_{\text{slash}}}{\ln(3)}$ be the stake factor for a single-file Sybil node.

The marginal gain in KOR rewards is: $\Delta r = \varepsilon_f(t) \cdot \left(\frac{|\mathcal{N}_f| - 1}{(|\mathcal{N}_f|)(|\mathcal{N}_f| + 1)}\right)$

The marginal cost (opportunity cost) is: $C_{\text{sybil,KOR}} = k_f \cdot \lambda_S \cdot \delta$

The attack is unprofitable when $\Delta r \leq C_{\text{sybil,KOR}}$, which requires:

$$\lambda_{\text{slash}} \geq \ln(3) \cdot \left(\left(\frac{r_k \cdot G_N}{\delta}\right) - 1\right) \tag{44}$$

where $r_k = \varepsilon_f \frac{t}{k_f}$ is the emission-to-stake ratio and $G_N = \frac{|\mathcal{N}_f| - 1}{(|\mathcal{N}_f|)(|\mathcal{N}_f| + 1)}$ is the geometry factor.

Numerical Example: With $\lambda_{\text{slash}} = 30$ and typical parameters ($\delta \approx 0.0000034$, $|\mathcal{N}_f| = 10$), the marginal gain (~155.7 KOR/block) is less than the marginal cost (~210.4 KOR/block), demonstrating sufficient security margin.

4.2.2.3 Crowding-Out Attack

An attacker creates Sybil nodes to dilute honest nodes' rewards and force them to exit.

Analysis: Each Sybil node requires stake $k_f \cdot \lambda_{\text{stake}}$ where $\lambda_{\text{stake}} \approx 28.30$ for single-file nodes. As the number of Sybil nodes k increases, per-node reward $\varepsilon_f \frac{t}{n+k}$ approaches zero while costs remain constant. The attack becomes increasingly unprofitable with scale. Honest nodes can simply wait—as the attacker bleeds capital, they will eventually exit.

4.2.2.4 Coordinated Departure Attack

Multiple colluding nodes coordinate to leave simultaneously, attempting to drive replication below $n_{\min}$.

Analysis: The leave fee increases quadratically as replication approaches $n_{\min}$:

$$\varphi_{\text{leave}}(f,t) = k_f \cdot \left(\frac{n_{\min}}{|\mathcal{N}_f|}\right)^2 \tag{45}$$

For $n = 5$, $n_{\min} = 3$, $k_f = 100$ KOR, sequential departures cost 36, 56.25, and 100 KOR (total: 192.25 KOR). The rapidly escalating costs and the hyperbolically increasing rewards for remaining nodes create strong disincentives.

4.2.2.5 Wash-Trading Attack

An attacker stores their own data through nodes they control to farm rewards.

Analysis: To store a file, the attacker pays a one-time fee υ_f which is entirely burned. The attack is profitable only if the NPV of future rewards exceeds this upfront cost. The profitability condition:

$$\left(\varepsilon(t_{\text{create}}) \cdot \frac{1-(1+\delta)^{-h}}{\delta}\right) > \chi_{\text{fee}} \cdot c_{\text{stake}} \cdot \ln\left(1 + \frac{|\mathcal{F}(t_{\text{create}})|}{F_{\text{scale}}}\right) \tag{46}$$

Setting χ_{fee} appropriately ensures wash-trading remains unprofitable.

4.2.2.6 File Size Manipulation Attacks

Small File Attack (Spam): Creating many tiny files to dilute rewards. The cost-to-influence ratio is constant regardless of file size (both cost and emissions scale with ln(size)), providing no leverage. A minimum file size $s_{\min}$ further mitigates spam.

Large File Attack: Creating massive files to capture disproportionate emissions. Both cost and emissions scale identically with ln(size), eliminating size-based leverage.

4.2.2.7 Data Gatekeeping Attack

A cartel of existing nodes refuses to share file data with newcomers to maintain a monopoly.

Mitigation: The protocol's sponsorship mechanism creates a competitive market that breaks cartels. Any cartel member has strong incentive to defect and capture sponsorship commissions. In equilibrium:

$$\gamma_{\text{eq}}(D) \approx \frac{c^{\text{USD}}_{\text{transfer}} \cdot \delta \cdot \left(|\mathcal{N}_f| + 1\right)}{\varepsilon_f(t) \cdot \xi_{\text{KOR/USD}} \cdot (1-(1+\delta)^{-D})} \tag{47}$$

For typical parameters, equilibrium commission rates are approximately 15%, making defection profitable.

4.2.3 Systemic Risks

4.2.3.1 Replication Collapse

If KOR price falls sufficiently, nodes may exit en masse, leaving files under-replicated.

Analysis: This is an accepted risk, not a bug. The protocol tolerates reduced redundancy when profitability falls. Files remain available as long as at least $n_{\min}$ nodes store them. The quadratic leave fee ($\varphi_{\text{leave}} = k_f \cdot \left(\frac{n_{\min}}{|N_f|}\right)^2$) makes exits increasingly expensive as replication approaches the minimum, creating a natural floor.

Residual risk: If KOR becomes worthless, the system fails. This is inherent to all cryptoeconomic systems—there is no protocol-level mitigation for complete token collapse.

5 Appendix

5.1 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.

5.1.1 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.

5.1.2 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 the Failure Detection section 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.

5.1.3 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: $\beta_{\text{bond}} \cdot \xi_{\text{KOR/USD}} \geq c_{\text{BTC}}^{\text{offer}} \cdot \xi_{\text{BTC/USD}} + c_{\text{transfer}}^{\text{USD}}$
- 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.

5.1.4 File Size Constraints

Limits:

- $s_{\text{min}} = 10$ KB - Minimum file size
- $s_{\text{max}} = 100$ 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.

5.1.5 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.

For cryptographic primitive definitions and related work, see the Kontor Proof-of-Retrievability.[3]

6. 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 and Alexey Gribov, "Kontor Proof-of-Retrievability," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/crypto

[4] Adam Krellenstein, Wilfred Denton, and Ouziel Slama, "Kontor: A New Bitcoin Metaprotocol for Smart Contracts and File Persistence," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/whitepaper

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