KONTOR PROOF-OF-RETRIEVABILITY

January 10, 2026

ADAM KRELLENSTEIN
adam@kontor.network

ALEXEY GRIBOV
alexey@kontor.network

Contents

1 Introduction 2
2 Multi-File Proof Aggregation 2
2.1 Public Input Structure 3
2.2 Proof Structure and Challenge Binding 4
2.3 Ledger Operational Details 5
3 Proof Generation (Nova IVC) 6
4 PoR Step Circuit φ_{PoR} 7
5 Proof Verification (Nova IVC) 9
6 Security Properties 10
6.1 Security Model 10
6.2 Computational Assumptions 10
6.3 Attack Resistance 11
6.4 Public Parameter Binding 12
6.5 Concrete Security Parameters 12
7 Cryptographic Primitives 12
8 Related Work 14
9. Bibliography 14

1 Introduction

This document specifies the cryptographic system underlying the Kontor Storage Protocol's proof-of-retrievability scheme. The system uses Nova[1] recursive SNARKs via the `arecibo`[2] library to generate compact proofs that can be efficiently verified (~50ms). The compressed SNARK is constant-size (~12 kB) regardless of the number of challenged symbols; the full proof includes per-file metadata that scales linearly with k files (~12 kB + 40 bytes $\times$ k).

The Kontor Storage Protocol is a system for ensuring that a set of untrusted actors continuously and correctly store data they have publicly committed to preserving. Storage nodes are challenged pseudo-randomly using a shared source of entropy (Bitcoin block hashes). With each block, indexers use this entropy to select file-node pairs for auditing. The chosen storage nodes must respond by publishing a proof demonstrating they possess a pseudo-random subset of the challenged file data. Nodes that fail to produce valid proofs within the allotted timeframe are subject to slashing of their escrowed KOR stake.

The cryptographic proof system addresses three key requirements:

Compact Proofs: Storage nodes are frequently challenged on multiple files within the proof submission window (W_{proof} blocks). To minimize Bitcoin transaction fees, the protocol allows nodes to aggregate challenges for multiple files into a single proof. The compressed SNARK is constant-size ($\approx$ 12 kB) regardless of the number of challenged symbols; the full proof includes per-file metadata (challenge IDs, ledger indices) that adds $\approx$ 40 bytes per file.

Efficient Verification: Proofs must be verified by all Kontor indexers deterministically. The Nova/Spartan construction enables verification in approximately 50ms, making on-chain verification practical at scale.

Soundness: Nodes must not be able to forge proofs without possessing the challenged file data. Merkle tree commitments verified within the SNARK circuit bind proofs to specific data, with computational soundness inherited from the discrete logarithm assumption over the Pallas/Vesta curve cycle.

For protocol context, economic incentives, and state machine specification, see the Kontor Storage Protocol.[3] For a high-level overview of the Kontor system as a whole, see the Kontor Whitepaper.[4]

2 Multi-File Proof Aggregation

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

Root Commitment: For each file f with Merkle root ρ_f and tree depth d_f, the protocol computes a root commitment using a domain-separated tagged Poseidon hash:

$$\text{rc}_f = \mathcal{H}_{\text{Poseidon}}(\text{TAG}_{\text{rc}}, \rho_f, d_f) \tag{1}$$

where TAG_{rc} is a fixed small integer constant embedded as a field element and $\mathcal{H}_{\text{Poseidon}}(\text{TAG}, x, y)$ denotes a single Poseidon sponge hash absorbing the three field elements (TAG, x, y) and outputting one field element. The root commitment binds together the file's Merkle root and its tree depth, preventing depth-spoofing attacks where an adversary might try to reuse proofs across files with different tree structures.

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

$$\mathcal{L} = \text{Merkle-Tree}\Big(\big[\!\big[\text{rc}_1, \text{rc}_2, ..., \text{rc}_{|\mathcal{F}|}\big]\!\big]\Big) \tag{2}$$

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

Aggregated Proof Structure: When a node is challenged on k files $\{f_1, f_2, ..., f_k\}$, it generates a single proof. Let s_{chal} denote the number of symbols challenged per file (a protocol parameter; see Kontor Whitepaper[4]). The uncompressed IVC witness contains:

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

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

The compressed SNARK (after Spartan compression) is constant-size (≈ 12 kB) regardless of k, $|\mathcal{F}|$, or the number of challenged symbols. Nova's IVC folding combined with Spartan's succinct verification enables this compression: the variable-size witness reduces to a constant-size SNARK that proves all symbol verifications and ledger inclusions were performed correctly. The full proof additionally includes per-file metadata (challenge IDs and ledger indices) adding ≈ 40 bytes per file.

2.1 Public Input Structure

The circuit uses a unified public input structure parameterized by k (files per step). Single-file mode ($k = 1$) is used when a prover submits a proof for exactly one file; multi-file mode ($k > 1$) is used otherwise, where k is the next power of two $\geq$ number of files in the proof. The public inputs are organized as:

$$\boldsymbol{z}_0 = \big[\rho_{\mathcal{L}}, s_0, \boldsymbol{I}_{\text{ledger}}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{L}_{\text{out}}\big] \tag{3}$$

where:

- $\rho_{\mathcal{L}}$ - Aggregated ledger root (in single-file mode, this is the file's Merkle root ρ directly, bypassing ledger lookup)
- s_0 - Initial state accumulator (0)
- $\boldsymbol{I}_{\text{ledger}} = [i_1, ..., i_k]$ - Ledger indices for each file slot (unused in single-file mode; 0 for padding slots)
- $\boldsymbol{D} = [d_1, ..., d_k]$ - Tree depths for each file slot (0 indicates a padding slot)
- $\boldsymbol{\Sigma} = [\sigma_1, ..., \sigma_k]$ - Challenge seeds for each file slot
- $\boldsymbol{L}_{\text{out}} = [\ell_1, ..., \ell_k]$ - Challenged leaf values for each file slot (public outputs)

The total arity is $2 + 4k$ field elements.

Leaf Outputs: The $\boldsymbol{L}_{\text{out}}$ values are the actual challenged leaf field elements (31-byte symbols encoded as field elements), not state accumulators. The circuit exposes these publicly so verifiers can confirm which symbols were proven. For padding slots (where $d_j = 0$), the leaf output is set to zero.

Padding Slots: When the number of challenged files is not a power of two, the remaining slots are filled with padding. Padding slots have $d_j = 0$, which the circuit uses to gate their processing: padding slots do not contribute to the state accumulator or require valid Merkle paths.

The circuit verifies that each real file's root commitment $\text{rc}_j = \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{rc}}, \rho_j, d_j)}$ is located at ledger index i_j in the tree with root $\rho_{\mathcal{L}}$, then verifies the symbol Merkle path for each challenged file using the per-file parameters. In single-file mode ($k = 1$), the ledger lookup is bypassed: the circuit directly checks that the computed file root matches $\rho_{\mathcal{L}}$, and the verifier confirms that this root matches the file's known Merkle root.

2.2 Proof Structure and Challenge Binding

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

$$\begin{aligned} \pi = \big(&\pi_{\text{compressed}}, \text{compressed SNARK} \\ &\textbf{challenge\_ids}, \text{ordered list of challenge identifiers} \\ &\rho_{\mathcal{L}}, \text{ledger root used for proof generation} \\ &\boldsymbol{I}_{\text{proof}}, \text{ledger indices at proof generation time} \\ &d_{\text{agg}} \text{ aggregated tree depth at proof generation}\big) \end{aligned} \tag{4}$$

The proof does not need to carry $\boldsymbol{z}_0$ or any intermediate/final IVC state explicitly. Verifiers reconstruct $\boldsymbol{z}_0$ deterministically from:

- the proof metadata $\big(\rho_{\mathcal{L}}, \boldsymbol{I}_{\text{proof}}, d_{\text{agg}}\big)$
- the retrieved challenge set (file roots, seeds, depths)

Ledger Indices in Proof: The proof includes $\boldsymbol{I}_{\text{proof}}$, the ledger indices for each file at proof generation time. This enables historical root validation: proofs generated against older ledger states remain valid because the SNARK proves the indices are correct for the claimed root $\rho_{\mathcal{L}}$. Verifiers validate that $\rho_{\mathcal{L}}$ is in their historical roots set (multi-file only), then use the proof's indices directly. Overhead is 8 bytes per file (fixed-width encoding); for typical proofs ($k \leq 10$ files), this adds < 100 bytes to the 12 KB compressed SNARK.

Challenge ID Ordering: The `challenge_ids` list is an explicit coverage set. Verifiers treat it as a list of IDs to retrieve the corresponding challenge objects and then sort challenges deterministically by the total order $((\text{file\_id}), (\text{challenge\_id}))$ before constructing per-file arrays for public inputs. This makes verification independent of any caller-provided list ordering, even if multiple challenges reference the same file.

Cross-Block Aggregation: Challenges from different block heights can be aggregated into a single proof. Each challenge carries its own seed σ derived from its originating block hash, enabling independent challenge derivation regardless of when the challenge was created. The prover selects a ledger root $\rho_{\mathcal{L}}$ (typically the current root) containing all challenged files and generates the proof against this state. Nodes can batch challenges across the entire proof submission window W_{proof}.

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

1. Extract the ordered `challenge_ids` list and ledger root $\rho_{\mathcal{L}}$ from the proof
2. If the proof is multi-file ($d_{\text{agg}} > 0$): verify that $\rho_{\mathcal{L}}$ is in the set of accepted historical roots
3. Retrieve each challenge $\mathcal{C}_j$ by ID from protocol state
4. Verify length: number of challenge IDs must match expected count
5. Reconstruct public inputs from challenges and proof-provided indices
6. Verify the SNARK against these deterministic public inputs

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

- Submitting a valid proof for different challenges (ID mismatch)
- Claiming to answer more/fewer challenges than proven (length mismatch)
- Using a fabricated ledger root (historical root validation)

Submitting an invalid proof is a slashable event: indexers treat it as an immediate failure of each referenced open challenge (see the Kontor Storage Protocol[3]).

Ledger State Binding: Proofs are cryptographically bound to a specific ledger state through the aggregated ledger root $\rho_{\mathcal{L}}$ included in the proof. The verifier checks that this

root appears in its set of accepted historical roots, ensuring the prover used a legitimate ledger state. This binding prevents:

- **Ledger substitution**: Cannot prove against a fabricated ledger with different file positions
- **Index manipulation**: Cannot claim files are at different ledger indices than their canonical positions

The combination of challenge ID ordering and historical root validation ensures that proofs are uniquely tied to specific challenges at a legitimate ledger state, preventing proof reuse or substitution attacks.

2.3 Ledger Operational Details

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

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

$$n'_{\text{ledger}} = \begin{cases} 1 & \text{if } |\mathcal{F}| = 0 \\ \text{next\_power\_of\_two}(|\mathcal{F}|) & \text{otherwise} \end{cases} \tag{5}$$

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

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

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

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

Indexer Caching: Indexers maintain the complete ledger tree structure:

- All leaf values (root commitments for each file in $\mathcal{F}$)
- All internal nodes (cached intermediate hashes)
- Sorted mapping from file identifier to ledger index

This full tree enables efficient Merkle path generation for challenge verification. The storage cost is $O(|\mathcal{F}|)$ field elements: for $|\mathcal{F}|$ files padded to n'_{ledger}, the tree contains n'_{ledger} leaves and $n'_{\text{ledger}} - 1$ internal nodes, totaling $\approx 2n'_{\text{ledger}}$ field elements. Since $|\mathcal{F}| \leq n'_{\text{ledger}} < 2|\mathcal{F}|$ and each field element is 32 bytes, this yields 64–128 $|\mathcal{F}|$ bytes. For a network with 1 million files, the ledger tree requires at most ≈ 128 MB of indexer storage.

Storage Node Caching: Storage nodes do not need to maintain the complete ledger tree. Instead, when generating proofs, nodes obtain:

- The current ledger root $\rho_{\mathcal{L}}$ from recent blockchain state
- Merkle paths for their challenged files from an indexer or by reconstructing locally

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

Historical Ledger Roots: Indexers maintain a set of accepted historical ledger roots to enable cross-block proof aggregation. When a prover generates a proof, they include the ledger root $\rho_{\mathcal{L}}$ used for proof generation. The verifier checks that this root is in the accepted set before validating the SNARK.

The accepted set contains roots from recent ledger states, covering at least the proof submission window W_{proof}. Specifically, indexers track:

- The current ledger root
- All roots from ledger updates (file activations) within the last W_{proof} blocks

This ensures any proof generated against a ledger state within the proof window will have its root accepted. Provers typically use the current ledger root, but may use slightly older roots if proof generation started before a recent file activation.

Storage Cost: Historical root tracking grows at one 32-byte entry per file activation. For a network activating 1000 files per day over a 2-week window (≈ 14000 activations), storage is ≈ 450 kB. This is negligible compared to the ledger tree itself (≈ 64 MB for 1M files).

3 Proof Generation (Nova IVC)

Important implementation note (Nova/Arecibo step semantics). In `arecibo`, the first call to `RecursiveSNARK::prove_step` after `RecursiveSNARK::new(...)` is a deliberate no-op: it increments the internal step counter but does not synthesize the circuit for that step. Therefore, to produce a proof with N logical PoR steps (i.e., N challenged-symbol iterations), an implementation must:

- Call `RecursiveSNARK::new(...)` once (this executes step 0 with synthesis).
- Call `prove_step(...)` exactly N times (not $N-1$). The first call is the no-op, and only calls $2..N$ synthesize steps $1..N-1$.
- Verify with `num_steps = N`.

Algorithm 1: Proof Generation (Nova IVC)

procedure NOVA.PROVE(challenges, node_storage, state)
  ▷ Node generates aggregated proof for k challenges
  ▷ Step 1: Determine circuit shape
  $k \leftarrow$ next_power_of_two(|challenges|)
  $d_{\max} \leftarrow \max_{\mathcal{C} \in \text{ challenges}} \text{depth}(\mathcal{C})$
  $d_{\text{agg}} \leftarrow$ aggregated_tree_depth(state)
  ▷ Circuit shape: $(k, d_{\max}, d_{\text{agg}})$

  ▷ Step 2: Build public inputs
  $\rho_{\mathcal{L}} \leftarrow$ state.get_ledger_root()
  $s_0 \leftarrow 0$
  ▷ Initialize per-file arrays (pad with zeros for unused slots)
  $\boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{L} \leftarrow$ arrays of length k
  for $j \in [0, k)$ **do**
    if $j <$ |challenges| **then**
      $\mathcal{C}_j \leftarrow$ challenges[j]
      $\boldsymbol{I}[j] \leftarrow$ ledger_index$(\mathcal{C}_j.\text{file_id}, \rho_{\mathcal{L}})$
      ▷ Compute index using current ledger root
      $\boldsymbol{D}[j] \leftarrow \text{depth}(\mathcal{C}_j)$
      $\boldsymbol{\Sigma}[j] \leftarrow \mathcal{C}_j.\text{seed}$
      $\boldsymbol{I}[j], \boldsymbol{D}[j], \boldsymbol{\Sigma}[j] \leftarrow 0, 0, 0$
      ▷ Padding slot
    end

end
$\boldsymbol{z}_0 \leftarrow [\rho_{\mathcal{L}}, s_0, \boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \mathbf{0}]$

▷ Step 3: Generate public parameters
pp $\leftarrow \mathcal{G}(k, d_{\max}, d_{\text{agg}})$
▷ Nova setup: Pallas/Vesta cycle with IPA commitment

▷ Step 4: Initialize Nova IVC accumulator
$\Pi_0 \leftarrow$ Nova.Init(pp, $\boldsymbol{z}_0$)

▷ Step 5: Iteratively prove N symbols per file
$N \leftarrow$ challenges[0].num_challenges
if $\exists \mathcal{C} \in$ challenges : $\mathcal{C}$.num_challenges $\neq N$ **then**
  return $\perp$ (mismatched iteration count)
end
current_state $\leftarrow s_0$
for $i \in [1, N]$ **do**
  ▷ Build witness for all k file slots
  $\boldsymbol{w}_i \leftarrow$ empty witness structure
  for $j \in [0, k)$ **do**
    if $\boldsymbol{D}[j] > 0$ **then**
      ▷ Real file: derive challenge and extract data
      $h \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge}}, \boldsymbol{\Sigma}[j], \text{ current\_state})}$
      if $k > 1$ **then**
        $h \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge\_per\_file}}, h, j)}$
      end
      $c_j \leftarrow$ derive_index_unbiased$\big(h, 2^{\boldsymbol{D}[j]}\big)$
      $\boldsymbol{w}_i$.file$[j] \leftarrow$ get_witness$\big($challenges$[j], c_j\big)$
      $\boldsymbol{w}_i$.file$[j] \leftarrow$ padding_witness()
    end
  end

  ▷ Execute step circuit and fold
  $\boldsymbol{z}_i \leftarrow \varphi_{\text{PoR}(\boldsymbol{z}_{i-1}, \boldsymbol{w}_i)}$
  $\Pi_i \leftarrow$ Nova.Fold$($pp$, \Pi_{i-1}, \boldsymbol{w}_i, \boldsymbol{z}_{i-1}, \boldsymbol{z}_i)$
  current_state $\leftarrow \boldsymbol{z}_{i[1]}$
end

▷ Step 6: Compress and package proof
$\pi_{\text{compressed}} \leftarrow$ Spartan.Compress(pp, Π_N)
proof $\leftarrow \big\{$challenge_ids : [challenges$[j]$.id for $j \in [0, |$challenges$|)]$, compressed_snark : $\pi_{\text{compressed}}$, ledger_root : $\rho_{\mathcal{L}}$, ledger_indices : $\boldsymbol{I}$, aggregated_tree_depth : $d_{\text{agg}}\big\}$
return proof
end

4 PoR Step Circuit φ_{PoR}

The PoR step circuit φ_{PoR} is executed within the Nova IVC fold operation. Each step processes k file slots (where k is the `files_per_step` parameter), verifying one challenged symbol per file and updating a single running state accumulator. Padding slots (identified by $d_j = 0$) are gated to have no effect.

Algorithm 2: PoR Step Circuit

procedure $\varphi_{\text{PoR}}(\boldsymbol{z}_{\in}, \boldsymbol{w})$
  ▷ Circuit executed inside Nova fold to verify symbols across k files
  ▷ Step 1: Parse public inputs
  $\rho_{\mathcal{L}}$ (aggregated root) $\leftarrow \boldsymbol{z}_{\in[0]}$
  s (state accumulator) $\leftarrow \boldsymbol{z}_{\in[1]}$

$\boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{L} \leftarrow$ per-file arrays from $\boldsymbol{z}_{\in}$
▷ Ledger indices, depths, seeds, leaf outputs for k slots

▷ Step 2: Process each file slot
for $j \in [0, k)$ **do**
  ▷ Extract per-file public inputs and witness
  $d_j \leftarrow \boldsymbol{D}[j]$
  $\sigma_j \leftarrow \boldsymbol{\Sigma}[j]$
  $\boldsymbol{w}_j \leftarrow \boldsymbol{w}$.file_witness$[j]$
  $\ell_j \leftarrow \boldsymbol{w}_j$.leaf

  ▷ Compute gating flag: slot is active iff $d_j > 0$
  $\text{gate}_j \leftarrow d_j > 0$
  ▷ Padding slots have $d_j = 0$ and are skipped

  ▷ Derive challenge index for this file
  $h_j \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge}}, \sigma_j, s)}$
  if $k > 1$ **then**
    ▷ Multi-file: mix challenge with file index
    $h_j \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{challenge\_per\_file}}, h_j, j)}$
  end

  ▷ Verify Merkle path (gated by gate_j)
  $\rho_j \leftarrow \text{verify\_merkle\_path\_gated}\big(\ell_j, \boldsymbol{w}_j.\text{siblings}, h_j, d_j, \text{gate}_j\big)$

  ▷ Compute root commitment and verify ledger membership
  $\text{rc}_j \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{rc}}, \rho_j, d_j)}$
  if $k > 1$ **then**
    Assert (gated)$(\text{rc}_j$ at index $\boldsymbol{I}[j]$ in tree $\rho_{\mathcal{L}})$
    Assert (gated)$(\rho_j = \rho_{\mathcal{L}})$
  end

  ▷ Conditionally update state accumulator
  $s' \leftarrow \mathcal{H}_{\text{Poseidon}(\text{TAG}_{\text{state}}, s, \ell_j)}$
  $s \leftarrow$ if gate_j then s' else s

  ▷ Set public leaf output (gated)
  $\boldsymbol{L}'[j] \leftarrow$ if gate_j then ℓ_j else 0
end

▷ Step 3: Build output state
$\boldsymbol{z}_{\text{out}} \leftarrow [\rho_{\mathcal{L}}, s, \boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{L}']$
▷ Carry forward root, indices, depths, seeds; update state and leaves
return $\boldsymbol{z}_{\text{out}}$
end

Circuit Properties:

- **Determinism:** All computations are deterministic given public/private inputs
- **Uniform Shape:** Circuit constraint count is fixed for a given $\big(k, d_{\max}, d_{\text{agg}}\big)$ shape, regardless of which slots are padding. This uniformity is essential for Nova's folding.
- **Completeness:** Honest prover with valid data always produces accepting proof
- **Soundness:** Prover without data cannot forge valid Merkle paths

Gating Mechanism: The circuit uses gated constraints to handle padding slots without changing the circuit shape. For each file slot j:

- The gating flag $\text{gate}_j = \big(d_j > 0\big)$ is computed from the public depth
- All assertions (Merkle path, ledger membership, root match) are gated: $\text{gate}_j \cdot (\text{computed} - \text{expected}) = 0$
- State updates and leaf outputs are conditionally selected based on gate_j

This ensures padding slots (with $d_j = 0$) satisfy all constraints trivially while real slots are fully verified.

5 Proof Verification (Nova IVC)

Algorithm 3: Proof Verification (Nova IVC)

procedure NOVA.VERIFY(proof, state)
  ▷ Indexer verification of storage proof
  ▷ Step 1: Validate challenge IDs and retrieve challenges
  ids ← proof.challenge_ids
  challenges ← empty list
  for id ∈ ids **do**
    $\mathcal{C} \leftarrow$ state.get_challenge(id)
    if $\mathcal{C} = \perp$ **then**
      return $\perp$ (challenge not found)
    end
    challenges.append($\mathcal{C}$)
  end

  ▷ Step 2: Validate ledger root (multi-file only)
  $\rho_{\mathcal{L}} \leftarrow$ proof.ledger_root
  $d_{\text{agg}} \leftarrow$ proof.aggregated_tree_depth
  if $d_{\text{agg}} > 0 \wedge \rho_{\mathcal{L}} \notin$ state.accepted_historical_roots() **then**
    return $\perp$ (invalid ledger root)
  end
  ▷ Ledger root validation is skipped for single-file proofs ($d_{\text{agg}} = 0$)

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

  ▷ Step 4: Reconstruct public inputs using proof metadata + challenges
  $\boldsymbol{I} \leftarrow$ proof.ledger_indices
  $\boldsymbol{D}, \boldsymbol{\Sigma} \leftarrow$ arrays from challenges, padded to length k
  $\boldsymbol{z}_0^{\text{expected}} \leftarrow [\rho_{\mathcal{L}}, 0, \boldsymbol{I}, \boldsymbol{D}, \boldsymbol{\Sigma}, \boldsymbol{0}]$

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

  ▷ Step 6: Verify SNARK
  $\pi_{\text{compressed}} \leftarrow$ proof.compressed_snark
  $N \leftarrow$ challenges[0].num_challenges
  if $\exists \mathcal{C} \in$ challenges : $\mathcal{C}$.num_challenges $\neq N$ **then**
    return $\perp$ (mismatched iteration count)
  end
  valid $\leftarrow$ Spartan.Verify$\left(\text{pp}, \pi_{\text{compressed}}, \boldsymbol{z}_0^{\text{expected}}, N\right)$
  ▷ Spartan verification: $O(\log^2 N)$ time, constant proof size
  if ¬ valid **then**
    return $\perp$ (proof verification failed)
  end

  return valid
end

6 Security Properties

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

6.1 Security Model

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

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

$$\Pr[\pi \leftarrow \mathcal{A}(\rho, c_1, ..., c_s) : \text{Verify}(\pi, \rho, c_1, ..., c_s) = \text{valid}] \leq \text{negl}(\kappa) \tag{6}$$

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

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

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

6.2 Computational Assumptions

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

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

Collision Resistance: The protocol assumes collision resistance for:

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

Collision resistance ensures that:

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

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

Transparent Setup: Unlike SNARKs such as Groth16 or PLONK[5], Nova and Spartan do not require a trusted setup ceremony. The public parameters can be generated by anyone deterministically from the circuit structure without relying on secret randomness that must be destroyed. This eliminates the need for multi-party computation ceremonies, removes the

risk of toxic waste compromise, and improves the protocol's decentralization properties. Any party can independently generate or verify the public parameters for a given circuit shape.

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

6.3 Attack Resistance

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

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

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

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

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

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

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

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

- High entropy: Block hashes have 256 bits of entropy from mining randomness
- Unpredictable: Cannot be predicted before mining completes
- Economically unbiasable: Miner grinding attacks are economically irrational (block rewards $>>$ challenge manipulation value)
- Independent: Each block's hash is independent of challenges

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

6.4 Public Parameter Binding

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

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

Parameter Flexibility: Different files may have different tree depths (different file sizes yield different symbol counts). The protocol supports this by:

- Allowing multiple parameter sets for different depths
- Using padding to standardize depths within a batch (all challenged files in single proof must have matching depths)
- Caching frequently used parameter sets to amortize generation costs

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

6.5 Concrete Security Parameters

Security Level: The protocol targets approximately 128-bit security. The bottleneck is the Pallas/Vesta curve cycle, which provides 126-bit security (254-bit prime fields). Other components meet or exceed this level:

- SHA-256: 128-bit collision resistance (256-bit output)
- Poseidon: Configured for 128-bit security over Pallas field

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

Field Sizes:

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

Proof Size: The compressed SNARK is approximately 12 kB regardless of the number of challenged symbols. The full proof includes per-file metadata (challenge IDs, ledger indices, ledger root) adding ≈ 40 bytes per file. For typical proofs ($k \leq 100$), total size is ≈ 16 kB; for large operators ($k = 1000$), total size is ≈ 52 kB.

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

7 Cryptographic Primitives

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

$$\Pr[(x, x') \leftarrow \mathcal{A}(1^\kappa) : x \neq x' \wedge \mathcal{H}(x) = \mathcal{H}(x')] \tag{7}$$

is negligible in κ, the security parameter.

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

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

$$\text{HKDF}_{\text{SHA256}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^\ell \tag{8}$$

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

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

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

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

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

- TAG_{leaf} (value 1) - Poseidon tag for leaf encoding (symbol-to-field-element conversion)
- TAG_{node} (value 2) - Poseidon tag for hashing internal Merkle tree nodes
- $\text{TAG}_{\text{challenge}}$ (value 6) - Poseidon tag for challenge index derivation and rejection sampling
- $\text{TAG}_{\text{state}}$ (value 7) - Poseidon tag for state accumulator updates
- TAG_{rc} (value 8) - Poseidon tag for root commitment computation (binds file root and depth)
- $\text{TAG}_{\text{challenge\_per\_file}}$ (value 9) - Poseidon tag for combining challenge with file index in multi-file proofs
- $\text{TAG}_{\text{challenge\_id}}$ (value 10) - SHA-256 domain tag for challenge identifier computation

Domain tags are small integer constants that prevent an adversary from constructing valid proofs by reusing hash outputs from different contexts. The tagged hash construction $\mathcal{H}_{\text{Poseidon}(\text{TAG}, x, y)} = \mathcal{H}_{\text{Poseidon}\left(\mathcal{H}_{\text{Poseidon}(\text{TAG}, x)}, y\right)}$ uses 2-arity Poseidon to build 3-input hashes.

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

- Leaves are padded to $n' = 2^{\lceil \log_2 n \rceil}$ (next power of two $\geq n$)
- Tree depth is $d = \log_2 n'$
- Each leaf node contains $\mathcal{H}(\text{TAG}_{\text{leaf}}, \ell_i)$
- Each internal node contains $\mathcal{H}(\text{TAG}_{\text{node}}, \text{left}, \text{right})$ where "left" and "right" are its children
- The root $\rho = \mathcal{T}.\text{root}$ commits to all leaves

A Merkle proof π_i for leaf ℓ_i is a path from leaf to root consisting of d sibling hashes, one per level.

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

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

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

8 Related Work

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

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

Cryptographic Foundations: The protocol leverages recursive proof composition via Nova[1] folding schemes, enabling efficient aggregation of multiple storage proofs into compact SNARKs. The implementation uses the Poseidon[7] hash function optimized for arithmetic circuits, Merkle trees[8] for cryptographic commitments, and Reed-Solomon[9], [17] erasure coding for fault tolerance. The proof system is built on Spartan[18] for transparent setup without trusted ceremonies.

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

9. 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, Alexey Gribov, and Ouziel Slama, "Kontor Storage Protocol," 2025. [Online]. Available: https://docs.kontor.network/docs/resources/storage-protocol

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

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

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

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

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

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

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

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

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

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

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

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

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

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