# ULI: Technical Specification

Timothy Bobinsky
timothy@uli.foundation
June 2026

**Abstract.** This document specifies the ULI (Unified Ledger Identity) protocol: a sovereign Layer-1 blockchain providing key-independent identity, Fast-HotStuff BFT consensus with BLS12-381 signature aggregation, a zero-knowledge credential system (Groth16/BN254 with Poseidon commitments), a privacy-layered social graph (social edges encrypted with per-edge AEAD; vouching topology intentionally public), and a capacity-staking economic model. Formal algorithm specifications, complexity analysis, performance bounds with empirical benchmarks, and a structured security analysis including threat model and catastrophic recovery scenarios are provided. For market context and economic rationale, see the companion whitepaper [22].

## 1. Introduction

This document is the formal technical specification of the ULI protocol. It defines the identity model, consensus mechanism, transaction format, social graph protocol, zero-knowledge credential system, economic model, moderation architecture, algorithms with complexity analysis, performance bounds with empirical benchmarks, and security analysis. For market context, use-case positioning, and economic rationale, see the companion whitepaper [22].

The three core separations (identity from keys, content from consensus, privacy from transparency) and design rationale are described in the whitepaper [22], Section 1. This document formalises the mechanisms that implement them.

The remainder of this specification describes the system model (Section 2), identity layer (Section 3), consensus protocol (Section 4), transaction format (Section 5), Anchored Social Graph Protocol (Section 6), zero-knowledge credential system (Section 7), economic model (Section 8), moderation (Section 9), formal algorithms and complexity analysis (Section 10), performance bounds with empirical benchmarks (Section 11), security analysis (Section 12), and related work (Section 13).

## 2. System Model

ULI is organised into three layers (see whitepaper [22], Figure 1 for the architecture diagram).

**Consensus Layer (L1).** A BFT state machine replicated across validators. It processes identity operations (account creation, key rotation, recovery, delegation), validator staking, schema registration, zero-knowledge credential verification, and hash-anchors for off-chain data. Content is *never* stored in consensus state; only its Blake3 hash enters the chain. The state is persisted in RocksDB with column families for accounts, keys, validators, schemas, nullifiers, and labels. All collections use `BTreeMap` (never `HashMap`) to ensure deterministic serialisation across validators. Serialisation uses Borsh; no floating-point arithmetic is permitted; all integer operations are checked.

**Data-Availability Layer.** Stores social-graph edges, post bodies, encrypted direct messages, and media references. Each piece of content is referenced on-chain by a `DaReference` and cryptographically bound by a Blake3 content hash. No content data enters the consensus state; only the hash-anchor transaction is replicated by validators.

**Blob addressing.** A `DaReference` is a tagged, content-addressed pointer: a 1-byte backend tag followed by a 32-byte Blake3 content hash and optional backend-specific metadata (max 256 bytes total). Backend tags distinguish the filesystem store (`0x02`) from the network-wrapped variant (`0x03`). The block header includes a `da_root`: a Merkle root of all DA content hashes in the block, enabling light-client verification of DA inclusion without downloading blobs.

**Erasure coding.** Blobs (max 2 MiB) are erasure-coded using Reed-Solomon over a finite field (FFT-based `reed-solomon-novelpoly` codec). The data is split into $K = \max(4, \lceil \text{len}/512 \rceil)$ chunks; $K$ parity chunks are generated for a default expansion factor of $N/K = 2$ (configurable up to 8). Any $K$ of $N$ chunks suffice to

reconstruct the original blob. Each chunk is hashed with Blake3 (domain-separated: `0x00 || data` for leaves, `0x01 || left || right` for internal nodes) and assembled into a Merkle tree whose root (`chunk_root`) is stored alongside the blob metadata. Merkle inclusion proofs for individual chunks are $O(\log N)$ hashes.

**Data Availability Sampling (DAS).** Validators do not download full blobs to verify availability. Instead, they perform probabilistic sampling: for each DA-bearing transaction, the validator requests 16 randomly selected chunks (seed = Blake3(blob_hash ‖ parent_hash), preventing proposer prediction), verifies each chunk against its Merkle proof, and accepts availability if all 16 samples pass. With a 2× expansion factor and 16 samples, the probability that an unavailable blob passes is $< 2^{-16} \approx 1.5 \times 10^{-5}$. If DAS fails, the validator withholds its BLS vote for the block; a quorum certificate therefore constitutes a collective attestation of data availability by $> 2S/3$ stake.

**Network protocols.** Blob dissemination uses two libp2p protocols:

- **Gossip** (`/uli/da-blobs`): full blob propagation via gossipsub (max 1 MiB per message).
- **Request-response** (`/uli/da-req/1`): retrieve a full blob by hash (32-byte request, length-prefixed response, max 2 MiB).
- **Chunk request** (`/uli/da-chunk/1`): retrieve a single chunk by (blob_hash, chunk_index), returning the chunk data plus Merkle proof siblings (max 64 KiB). This protocol serves DAS sampling requests.

**Consensus integration.** Before voting on a proposed block, each validator checks DA availability for every DA-bearing transaction: (1) extract the `DaReference`; (2) if DAS is enabled, run the 16-sample probabilistic check; otherwise, retrieve the full blob; (3) if the blob is unavailable, retry once after 2 s; (4) if still unavailable, refuse to vote and log a warning. During block proposal, the leader filters the mempool and excludes transactions whose DA references are locally unavailable. After consensus commit, blobs are marked as finalized at the committed block height for retention management.

**Retention and garbage collection.** Three retention policies are supported: `ProviderGuaranteed` (indefinite), `IncentivizedReplication{min_replicas}` (incentive-based), and `UserPaid{expiry_epoch}` (time-bounded). Garbage collection is finality-aware: blobs are eligible for deletion only after `da_retention_blocks` (default 50,400 ≈ 7 days) have elapsed since finalization. Pinned blobs are exempt from GC. A fisherman challenge protocol (where any participant may challenge a provider to produce a referenced blob, with failure resulting in stake slashing) is a candidate mechanism for economic enforcement of continued storage. However, on-chain proof of withholding faces the well-known fisherman's dilemma: a challenger cannot prove that the provider *never* served the data, only that it was not served to the challenger [23]. The protocol therefore does not rely on fisherman challenges for its security model; continued availability is ensured through erasure coding, validator chunk sampling, and provider diversity (Section 12.6). Fisherman challenges remain a future-work hardening measure.

**Long-term storage economics.** The default GC window (≈7 days) is a consensus-layer minimum, not a content lifetime. Social-graph data (posts, edges, DMs) requires persistent storage well beyond 7 days; the retention policy determines who bears this cost. `ProviderGuaranteed` shifts storage cost to the provider's infrastructure budget (not charged in CC or ULI). A provider serving 100K users with ≈10 GB incurs standard cloud costs (≈$200/month), outside the protocol's fee model. The protocol does not subsidise long-term storage; providers who sponsor accounts are responsible for persisting data. Providers who cease operation leave data subject to GC; users can export via `PortableProfile` (Section 6.8).

**Backend interface.** The DA backend implements the `DaClient` trait (`store`, `retrieve`, `is_available`, `batch_store`, `health_check`), providing a uniform interface for the consensus layer regardless of storage backend.

**Indexers.** Stateless read replicas that consume the L1 block stream and DA blobs, reconstruct the full social graph in a relational database (Postgres), and expose an HTTP API. All client reads go through indexers; the L1 never serves queries directly. Multiple independent indexers may operate concurrently, each implementing its own ranking and filtering algorithms. Indexers are not trusted for writes: all mutations flow through signed transactions submitted to L1.

Content anchors (`AnchorContent`, `SendDm`, `AnchorGraph`) are consensus transactions, but each carries only a hash and DA reference (~128–320 bytes), not the content itself. Throughput analysis is provided in Section 11.

## 3. Identity

### 3.1 ULI ID

Each account is identified by a ULI ID: a monotonically increasing `u64` counter assigned at account creation. The identifier never changes, regardless of key rotation, recovery, or device migration. Each ULI ID resolves to a W3C-compatible DID: `did:uli:<id>`. The DID document exposes the account's active public key, recovery configuration, and service endpoints, enabling interoperability with W3C Decentralized Identifiers [20] and enterprise identity wallet ecosystems. No personally identifiable information (email, phone, government ID) is stored on-chain.

**Definition 1** (Identity-Key Separation)**.** *Let $\mathcal{I}$ denote the identity space and $\mathcal{K}$ the key space. A ULI ID $i \in \mathcal{I}$ maps to a key registry $R(i) = (k_{active}, K_{retired})$ where $k_{active} \in \mathcal{K}$ and $|K_{retired}| \leq 16$. The mapping $R$ is mutable: key rotation replaces $k_{active}$ and appends the old key to $K_{retired}$ without altering $i$.*

The identity-key separation resolves a class of problems that plague key-based identity systems: key compromise does not require social-graph reconstruction, key rotation does not break existing references, and multiple devices can share identity through derived keys rather than sharing a single private key.

### 3.2 Key Management

Account keys use Sr25519 (Schnorr signatures on Ristretto255 [10]), chosen for direct compatibility with FROST-Ristretto [6] threshold signing. Sr25519 provides 128-bit security, is non-malleable, and avoids the cofactor issues of raw Ed25519 through the Ristretto encoding.

The key registry maintains a single active key and up to 16 retired keys, each annotated with the epoch at which it was retired. Rotation to a previously retired key is prohibited to prevent replay of old authorisation proofs. A reverse index maps each public key to its account ID, enabling $O(1)$ lookup during transaction validation.

### 3.3 Derived Keys

An account holder may authorise up to 32 derived keys, each scoped by a `SpendingLimit`:

- **Permission mode**: AllowList (only listed operations permitted) or DenyList (all except listed).
- **Global token limit**: maximum aggregate token expenditure.
- **Per-operation quotas**: a `BTreeMap<String, u64>` mapping operation types to invocation caps.
- **Expiration block**: the derived key becomes invalid after this height.

Derived keys enable applications to act on behalf of users without holding the owner key, analogous to OAuth scopes but enforced at the consensus layer. The account holder may revoke any derived key at any time via `RevokeDerivedKey`.

### 3.4 Agent and Bot Attestation

The protocol defines three account categories, distinguished by the credentials and delegation chains attached to each account:

1. **Human-attested.** The account holds a valid personhood credential (Section 7, circuit $\tau = 1$) issued by an external verifier. Applications that require proof of humanity check this credential via ZK proof; the protocol never learns *which* human the account represents.
2. **Bot-declared.** The operator self-identifies the account as automated. Declaration is a voluntary, on-chain attestation. Undeclared bots are not punished by the protocol; detection and response are delegated to the application and indexer layers.
3. **Delegated agent.** The account operates under a derived key issued by a parent account, with a spending limit that constrains which operations the agent may perform, how many times, and at what total cost. The delegation chain is auditable on-chain: any observer can verify that agent $A$ acts on behalf of account $B$ with permissions $P$.

The `SpendingLimit` structure provides fine-grained authorization:

- **AllowList mode**: only explicitly listed operations are permitted; all others are denied.
- **DenyList mode**: all operations except listed ones are permitted.
- **Token budget**: a global cap on aggregate token expenditure.
- **Per-operation caps**: individual invocation limits per operation type.
- **Expiration**: the delegation expires at a specified block height.

This mechanism serves as a cryptographic authorization layer for AI agents: an AI platform issues a derived key to each agent instance with an AllowList of permitted actions and a token budget, providing auditable, revocable, scope-limited authority without shared secrets. The delegation is enforced at the consensus layer, not by the agent's software.

### 3.5 Account Structure

The full account state maintained in consensus is:

- ULI ID (immutable `u64` counter).
- Created epoch.
- Key registry: one active Sr25519 key, up to 16 retired.
- Threshold config: optional FROST group key, threshold, total participants, DKG transcript hash.
- Recovery config: optional social or timelock configuration.
- Derived keys: `BTreeMap` of up to 32 grants.
- Credentials: up to 64 `CredentialCommitment` entries, each containing credential type (max 128 chars), Poseidon commitment, verification key hash, issued epoch, and optional expiration epoch.
- Reputation: up to 64 soulbound attributes (non-transferable, protocol-issued).
- Graph root: Blake3 Merkle root of the account's edge set.
- ZK graph root: Poseidon Merkle root for ZK proofs over graph structure.
- Disclosure policies: up to 32 policies controlling which credentials are revealed to which applications.
- Nonce: monotonically increasing transaction counter.
- Sponsor: optional provider ULI ID for capacity-sponsored accounts.

All fields except ULI ID are mutable through appropriate transactions. The account is the atomic unit of state in the consensus layer; all identity operations are scoped to a single account.

### 3.6 Recovery

Three complementary recovery mechanisms are available, each designed to prevent instant account takeover while remaining practical for legitimate recovery (Figure 1).

**Social Recovery.** The account holder designates up to 8 guardians (other ULI IDs) and a threshold $t$ where $1 \leq t \leq |\text{guardians}|$. Recovery requires $t$ guardian approvals followed by a mandatory timelock of 1,008,000 slots (approximately 7 days at 600 ms per slot). Cancellation behaviour during the waiting period is governed by a per-account policy (see Section 12.4). Guardian-set changes (`SetSocialRecovery`) are themselves subject to a timelock, preventing an attacker from silently removing guardians before initiating recovery.

**Timelock Recovery.** A pre-registered backup key may initiate recovery subject to a minimum waiting period of 432,000 slots (approximately 3 days). The same per-account cancellation policy applies.

**Threshold Recovery.** FROST-based [6] threshold signing: the account registers a group public key, threshold $t$, total participants $n$, and a DKG transcript hash. Recovery requires a valid group signature from any $t$-of-$n$ key-share holders. No timelock is imposed, as the threshold itself provides security proportional to the number of honest participants.

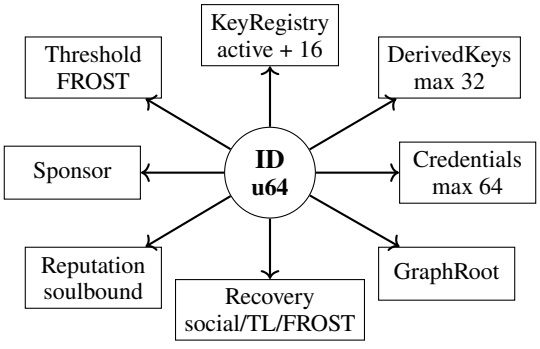


Figure 1: Account model. A ULI ID is the immutable root; all components (keys, credentials, graph, recovery, reputation) are mutable and independently upgradeable.

Figure 2 illustrates the complete account lifecycle from creation through social recovery execution.

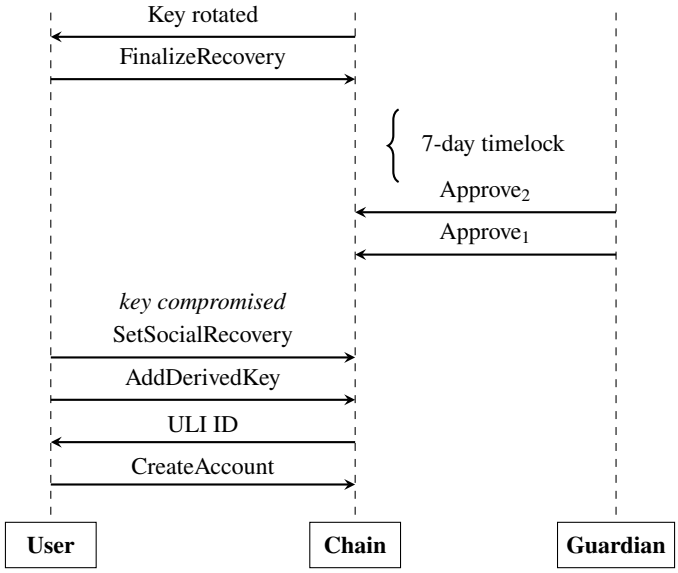


Figure 2: Account lifecycle: creation, key derivation, social recovery setup, and recovery execution with 7-day timelock.

Figure 3 compares the three recovery paths and their respective timelocks.

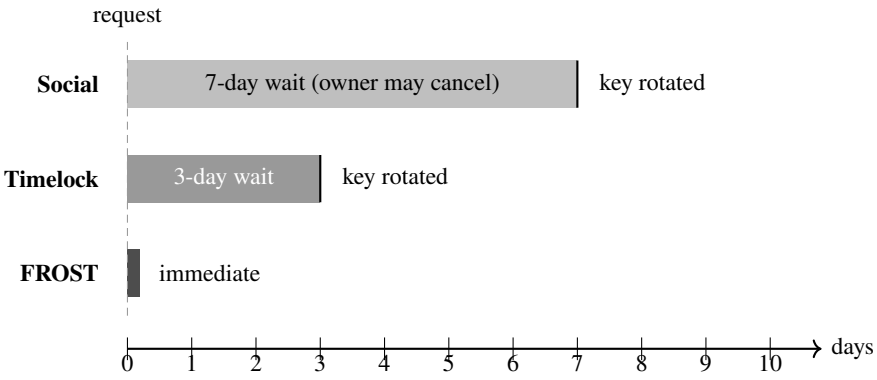


Figure 3: Recovery mechanism comparison. Social recovery requires the longest timelock (7 days); FROST threshold recovery is immediate but requires $t$-of-$n$ key shares.

## 4. Consensus

ULI uses Fast-HotStuff [2], a refinement of the original HotStuff protocol [1] that reduces the commit rule from 3 phases to 2 while preserving $O(n)$ message complexity per view. Vote aggregation uses BLS12-381 [3] threshold signatures.

### 4.1 Protocol Overview

The protocol operates in a sequence of numbered *views*. In each view, a deterministic leader proposes a block containing up to 256 transactions, validators verify the proposal and vote, and votes are aggregated into a quorum certificate (QC) via BLS signature aggregation.

**Definition 2** (Quorum)**.** *Given total stake $S$ across $n$ validators, the quorum threshold is*

$$q = \left\lfloor \frac{2S}{3} \right\rfloor + 1. \tag{1}$$

*A QC is valid when the aggregate BLS signature covers validators whose combined stake reaches $q$. This is the smallest integer strictly exceeding $2S/3$, which is necessary for the quorum intersection argument: any two sets of size $> 2S/3$ overlap in $> S/3$ stake, exceeding the Byzantine budget.*

The implementation computes this as $q = \lfloor 2S/3 \rfloor + 1$ using 128-bit intermediate arithmetic to prevent overflow. Leader election is stake-weighted and deterministic:

$$\text{seed} = H(v \,\|\, \text{qc.hash}), \tag{2}$$

where $v$ is the view number, $H$ is Blake3, and qc.hash is the hash of the highest QC known to the proposer. The seed selects a leader by cumulative-stake binary search over the active validator set sorted by identifier. Incorporating the QC hash prevents an adversary from predicting leaders far in advance.

### 4.2 2-Chain Commit Rule

ULI employs a 2-chain commit rule (Figure 4):

**Definition 3** (2-Chain Commit)**.** *Block $B_1$ proposed at view $v_1$ is committed when there exists a block $B_2$ at view $v_2 = v_1 + 1$ such that $B_2$ extends $B_1$ and carries a valid QC for $B_1$:*

$$\mathit{commit}(B_1) \iff \exists\, B_2 : B_2.\mathit{parent} = B_1 \;\wedge\; v_2 = v_1 + 1. \tag{3}$$

The consecutive-view requirement is critical: it ensures that no equivocating block could have been certified between $B_1$ and $B_2$. If there is a view gap ($v_2 > v_1 + 1$), the commit does not trigger.

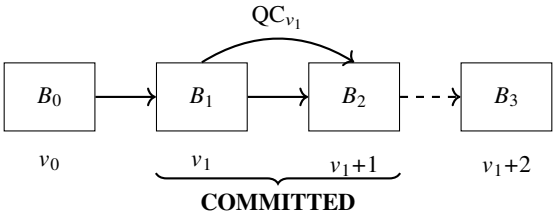


Figure 4: 2-chain commit rule. $B_1$ commits when $B_2$ at the consecutive view $v_1 + 1$ carries a valid QC for $B_1$.

### 4.3 Liveness

If a leader fails to produce a valid proposal within the view timeout, validators broadcast timeout votes. A *timeout certificate* (TC) aggregates $q$ timeout votes (by stake) and advances the view without committing a block. The protocol guarantees liveness under partial synchrony: after the Global Stabilisation Time (GST), an honest leader will produce a block that achieves a QC within bounded time.

View synchronisation handles network partitions: if a validator observes a QC from a view ahead of its own, it advances (subject to a maximum jump of 256 views to prevent DoS via fabricated high-view certificates).

### 4.4 Equivocation Slashing

If a validator signs two different blocks in the same view, any observer may submit an `SubmitEquivocationEvidence` transaction containing both BLS signatures. Both signatures are verified against the validator's registered key. Upon confirmation:

$$\text{slash}(s_v) = \left\lfloor \frac{s_v \cdot 1000}{10000} \right\rfloor = \left\lfloor \frac{s_v}{10} \right\rfloor, \tag{4}$$

where $s_v$ is the validator's total stake. The slash uses 128-bit intermediate arithmetic and is capped at $s_v$. The validator is jailed for a minimum of 7 epochs (approximately 21 days). Delegators to a slashed validator lose a proportional share, creating incentive to monitor validator behaviour and diversify delegations.

### 4.5 Safety

Each validator maintains a persistent `SafetyState`:

- `last_voted_view`: monotonically increasing; the validator refuses to vote at any view ≤ this value.
- `locked_qc`: the highest QC for which the validator has voted.
- `highest_qc`: the highest QC observed.

A validator votes for a proposal at view $v$ only if $v >$ last_voted_view and the proposal's justify QC is at least as high as the locked QC.

**Theorem 1** (Safety)**.** *If $n \geq 3f + 1$, where $f$ is the maximum number of Byzantine validators by stake, then no two honest validators commit different blocks at the same height.*

*Proof sketch.* Suppose for contradiction that honest validators commit $B$ and $B'$ at height $h$ with $B \neq B'$. By the 2-chain rule, $B$ was committed via a QC at view $v + 1$ extending a QC at view $v$; similarly $B'$ via views $v'$ and $v' + 1$. Each QC requires $q > 2S/3$ stake. Since honest validators hold $> 2S/3$ stake (by assumption $f < S/3$), two quorums must overlap in at least one honest validator. If $v = v'$, this validator voted for both $B$ and $B'$ at view $v$, contradicting the monotonic `last_voted_view` invariant. If $v < v'$, the validator locked on a QC at view $v$ before voting at $v + 1$, and later voted at view $v'$ for a block that does not extend $B$; but the locked QC at view $v$ prevents this, a contradiction. □

## 5. Transactions

### 5.1 Transaction Format

A signed transaction consists of a payload, signer public key (Sr25519), signature, nonce, and chain identifier. The signing message is:

$$m = \text{borsh}(\text{payload}) \,\|\, \text{nonce}_{\text{LE}} \,\|\, \text{chain\_id}_{\text{LE}}, \tag{5}$$

where LE denotes little-endian encoding. Including the nonce and chain ID prevents replay within a chain (monotonic nonce) and across chains (chain ID: mainnet = 1, devnet = $2^{32} - 1$, testnet = 0).

The transaction hash includes the signer for mempool deduplication:

$$H_{\text{tx}} = \text{Blake3}(\texttt{"uli-tx-v1"} \,\|\, m \,\|\, \text{signer}). \tag{6}$$

### 5.2 Transaction Types

Table 1 summarises the 49 transaction types across twelve categories.

### 5.3 Block Structure

A block consists of a header and up to 256 transactions. The header contains: height, parent hash, transactions Merkle root, state root, DA root, slot, epoch, proposer ULI ID, quorum BLS signature, and signer bitmap. The quorum signature and bitmap are excluded from the block hash (added post-vote). Block hash domain: `"uli-block-header-v1"`.

The transactions Merkle root uses domain-separated Blake3:

$$\begin{aligned} \text{leaf}(tx) &= \text{Blake3}(D \,\|\, \texttt{0x00} \,\|\, tx.\text{hash}), \\ \text{node}(l, r) &= \text{Blake3}(D \,\|\, \texttt{0x01} \,\|\, l \,\|\, r), \end{aligned} \tag{7}$$

where $D$ = `"uli-tx-merkle-v1"`. Domain bytes `0x00` (leaf) and `0x01` (node) prevent second-preimage attacks. Odd leaf counts: the last leaf is duplicated. Empty blocks have transactions_root = `Hash::ZERO`.

### 5.4 Time Model

Time is measured in slots of 600 ms. Epochs group 432,000 slots (approximately 3 days):

$$e(s) = \left\lfloor \frac{s}{432{,}000} \right\rfloor. \tag{8}$$

An epoch boundary occurs at slot $s$ when $(s + 1) \bmod 432{,}000 = 0$.

Table 2 summarises consensus-critical parameters.

Table 1: Transaction types by category (49 total).

| Category | Types |
|---|---|
| Identity | CreateAccount, RotateKey |
| Recovery | SetSocialRecovery, SetTimelockRecovery, ApproveSocialRecovery, InitiateTimelockRecovery, CancelRecovery, FinalizeRecovery, FinalizeGuardianChange, CancelGuardianChange |
| Delegation | AuthorizeDerivedKey, RevokeDerivedKey |
| Threshold | RegisterThresholdKey, ThresholdRecovery |
| ZK Credentials | AddCredential, RevokeCredential, VerifyCredentialProof, RegisterCircuit, DeprecateCircuit, AnchorAnonymousContent |
| Capacity | RegisterProvider, SponsorAccount, DeactivateProvider, ClaimProviderUnbonded, MintCapacityCredits, UpdateOraclePrice |
| Staking | RegisterValidator, DelegateStake, UndelegateStake, Unjail, ClaimUnbonded, SubmitEquivocationEvidence |
| Social | AnchorGraph, AnchorContent |
| Messaging | PublishDmKey, SendDm |
| Schemas | RegisterSchema, SchemaOperation |
| Indexing | RegisterIndexer, PublishScoreRoot, ChallengeScoreRoot |
| Moderation | RegisterLabeler, DeactivateLabeler, ApplyLabel, SubscribeLabeler, UnsubscribeLabeler, FileAppeal |
| Governance | SubmitProposal, CastVote |

Table 2: System parameters.

| Parameter | Value |
|---|---:|
| Max transactions per block | 256 |
| Max ZK transactions per block | 32 |
| Max credentials per account | 64 |
| Max derived keys per account | 32 |
| Max retired keys | 16 |
| Max guardians (social recovery) | 8 |
| Max unbonding entries | 32 |
| Max labeler subscriptions | 64 |
| Slot duration | 600 ms |
| Slots per epoch | 432,000 |
| Social recovery timelock | 1,008,000 slots |
| Min timelock recovery | 432,000 slots |
| Min jail duration | 7 epochs |
| Min provider stake | 1,000 tokens |
| Provider grace period | 7 epochs |
| Min validator self-stake | 10,000 tokens |
| Capacity per token | 10 units |
| Oracle TWAP window | 7 epochs |
| Labeler deposit | 10,000 tokens |
| Governance quorum | 33% |
| Governance threshold | 67% |
| Equivocation slash | 10% |

## 6. Anchored Social Graph Protocol

The Anchored Social Graph Protocol (ASGP) is the data-layer protocol governing how social relationships, content, and messages are stored, anchored, and verified in the ULI ecosystem. ASGP separates the social graph into off-chain storage (DA layer) and on-chain anchoring (Merkle roots), providing both scalability and cryptographic verifiability without burdening consensus with bulk data.

The protocol defines five operations:

1. `AnchorGraph`: update an account's edge-set Merkle root.
2. `AnchorContent`: bind a content hash and schema to a DA reference.
3. `RegisterSchema`: define a new content or edge type.
4. `PublishDmKey`: register a DM encryption public key.
5. `SendDm`: anchor an encrypted direct message.

Each operation produces a consensus-level transaction that modifies on-chain state (a Merkle root, a schema entry, a DM key) while the corresponding bulk data (edges, post bodies, ciphertext) lives on the DA layer and is retrieved by indexers.

### 6.1 Graph Structure and Dual Merkle Roots

Social-graph edges (follows, blocks, mutes, vouches, list memberships, and any schema-defined relationship type) are stored on the DA layer as typed records. An edge record contains: source account ID, target account ID, edge type (schema-defined), timestamp, and an optional encrypted payload.

Each account maintains two Merkle roots on-chain, updated atomically by `AnchorGraph`:

- `graph_root`: a Blake3 Merkle root over the account's current edge set. Used for data integrity verification: an indexer can prove that it is serving the correct edge set by providing a Merkle proof against this root.
- `zk_graph_root`: a Poseidon Merkle root over the same edge set, computed using the ZK-friendly hash. Used for zero-knowledge proofs over graph structure: a user can prove "I follow at least $N$ accounts," "account $X$ is in my graph," or "I have $M$ vouches from accounts with trust score above threshold $T$" without disclosing the complete edge set.

The dual-root design avoids forcing consensus to use a ZK-friendly hash (Poseidon is slower than Blake3 for non-circuit computation) while providing the in-circuit provability that ZK credentials require. The `AnchorGraph` transaction updates both roots atomically, ensuring consistency.

When a user follows or unfollows another account, the client: (1) computes the new edge set locally, (2) builds both Merkle trees, (3) publishes the updated edges to the DA layer, (4) submits an `AnchorGraph` transaction with the new roots and DA reference.

Edge types are schema-defined, allowing permissionless extension. The protocol does not prescribe a fixed set of relationship types; any registered schema may define new edge semantics (e.g., "endorses," "delegates-to," "blocks-for-reason").

### 6.2 Social Vouching and Sybil Resistance

The ASGP graph supports a vouching mechanism for building social trust. Users create weighted vouch edges to attest to another participant's personhood. These edges are stored encrypted on the DA layer and anchored via the ZK graph root.

Graph-based Sybil resistance, as studied by Shahaf et al. [11] in the context of BrightID, relies on the observation that Sybil accounts form weakly connected clusters at the periphery of the honest social graph. Algorithms such as SybilRank and its variants assign trust scores based on graph topology: accounts with dense connections to the honest core receive high scores, while Sybil clusters receive low scores.

ASGP improves on this model in two ways. First, a user can prove their trust score via the MerkleInclusion or SelectiveDisclosure ZK circuits without revealing which specific accounts vouched for them. Second, unlike BrightID's fully public graph, ASGP encrypts all social edges (Section 6.4), limiting topology exposure to the vouching subgraph (see below).

BrightID's experience also reveals the limitations of graph-based approaches: small-scale Sybil attacks (2–3 fake identities with disjoint verification groups) remain difficult to detect through topology alone [11]. Collusion among a small group of verified humans who vouch for fake accounts produces subgraphs indistinguishable from legitimate social connections.

ASGP does not ossify a specific Sybil-detection algorithm in consensus. Instead, it provides the cryptographic substrate (encrypted edges, ZK-provable trust scores, per-edge compartmentalised encryption) and delegates the scoring policy to *staked indexers*.

**Staked indexer commitments.** An indexer registers on-chain via a `RegisterIndexer` transaction with a bond $B_I \geq B_{\min}$ (governance-set, default 10,000 ULI). The registration includes an algorithm identifier (name, version, parameter set) that uniquely specifies a deterministic scoring function. At each epoch $e$, the indexer computes trust scores over the public vouching topology using a consensus-derived seed $\text{seed}_e = \text{Blake3}(\text{epoch} \| \text{parent\_hash})$ (eliminating non-determinism in algorithms like Louvain), constructs a Poseidon Merkle tree of $(account\_id, \tau)$ pairs, and publishes the root via a `PublishScoreRoot` transaction.

Any party may re-execute the published algorithm over the same input graph (vouching topology is public) and the same seed, and submit a `ChallengeScoreRoot` transaction containing a divergent root. Validators verify the challenge by re-executing the algorithm (bounded by $O(|V| + |E|)$ per epoch); if the challenge succeeds, the indexer's bond is slashed and the epoch root is invalidated. If the challenge fails, the challenger forfeits a challenge deposit ($0.1 \times B_{\min}$), preventing griefing.

ZK proofs of trust scores (MerkleInclusion circuit, $\tau = 3$) reference a specific indexer's on-chain root, making the trust chain explicit: the proof attests "my score exceeds $\theta$ according to indexer $I$, whose root is on-chain and whose bond backs its correctness."

Different indexers may register different algorithms; applications choose which indexer's scores to accept, creating a competitive market for Sybil detection anchored by staked commitments rather than unverifiable off-chain computation.

Sybil detection algorithms are upgradeable by registering new indexers without protocol changes.

### 6.3 Graph Privacy Model

The social graph has two privacy tiers, reflecting the distinct functions of vouching and social connection.

**Vouching edges (public topology, encrypted payload).** Vouching is a deliberate public act: when account $A$ vouches for account $B$, the pair $(A, B)$ is published in cleartext on the DA layer. The vouch payload (weight, context, timestamp) is encrypted per-edge via the ECIES scheme described below. Public vouching topology is necessary for trust scoring: indexers compute bridging coefficients (Definition 8) and trust scores (Algorithm 2) over the vouching subgraph at epoch boundaries. This is analogous to PGP key signing: the act of vouching is a public signal whose value depends on observability.

**Social edges (fully encrypted).** All other edges (follow, block, mute, direct messages) are encrypted independently using the ECIES scheme below. Source, target, edge type, and timestamp are all inside the encrypted payload. Indexers see only the account's Merkle root and DA pointer; they cannot reconstruct the social topology without the recipient's private key.

**ZK bridge between tiers.** Verifiers (banks, applications) never see either tier directly. A user proves trust properties via the MerkleInclusion ZK circuit against the Poseidon graph root, without revealing who vouched or the underlying topology. The public vouching graph is visible to indexers who compute the scores; the ZK proof hides the graph from the entities that consume the scores.

**Metadata side channels.** On-chain `AnchorGraph` transactions reveal which account updated its graph root and when, but not the content, targets, or number of edges. Batching multiple edge updates into a single anchor reduces the timing signal. `VerifyCredentialProof` transactions present a stronger concern: the nullifier hides the prover, but the sponsoring provider is visible through capacity, and the timestamp is on-chain. The pattern "provider $X$ verifies $N$ credentials per day" is observable, constituting potential business-data leakage for privacy-sensitive deployments. Mitigations include distributing volume across multiple provider accounts and batching proof submissions; these are operational measures, not protocol-level guarantees.

### 6.4 Per-Edge Encryption

Unlike broadcast-key systems that share a single decryption key across all followers, ASGP encrypts each edge and each direct message independently using an ECIES-style ephemeral key exchange (Figure 5):

1. The sender generates a fresh ephemeral X25519 [9] key pair $(e, e \cdot G)$ for each message or edge.
2. The ephemeral secret and the recipient's static DM public key $pk_B$ are combined via Diffie-Hellman: $\text{shared} = e \cdot pk_B$.
3. A domain-separated KDF derives the symmetric key from the shared secret. The domain tag distinguishes edge from DM encryption.
4. The derived key encrypts the payload with XChaCha20-Poly1305 (AEAD) using a random 24-byte nonce.

5. The ephemeral public key $e \cdot G$ is sent alongside the ciphertext; the ephemeral secret $e$ is discarded.

This provides *forward secrecy*: a fresh ephemeral key per message means compromising the recipient's long-lived key does not reveal past messages (ephemeral secrets are discarded). No Double Ratchet is used: ASGP's asynchronous store-and-forward model precludes the synchronised state a ratchet requires.

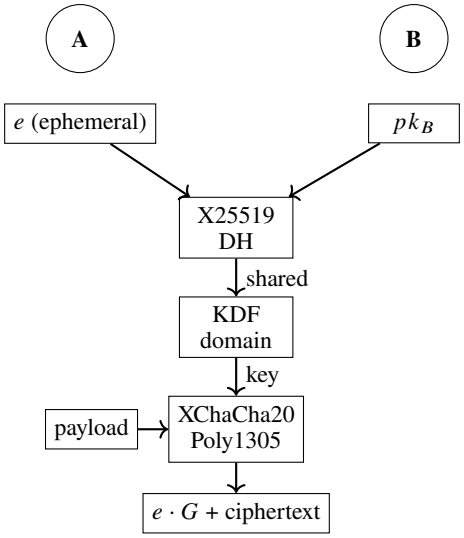


Figure 5: Ephemeral-static ECIES encryption. For each edge or DM, the sender generates a fresh ephemeral key $e$, performs DH with the recipient's static key $pk_B$, derives a symmetric key via domain-separated KDF, and encrypts with XChaCha20-Poly1305. The ephemeral secret is discarded after encryption, providing forward secrecy.

### 6.5 Content and Schemas

Content (posts, media, articles) is stored on the DA layer and anchored on-chain via `AnchorContent` transactions containing the content hash, schema identifier, DA reference, and retention policy.

Schemas are permissionlessly registered via `RegisterSchema` with a namespace, name, definition, and anti-spam deposit. Unlike DSNP/Frequency [8], no governance vote is required. Schema ID 0 denotes untyped content and is always valid.

Figure 6 illustrates the content lifecycle from DA upload through on-chain anchoring, indexing, and labeling.

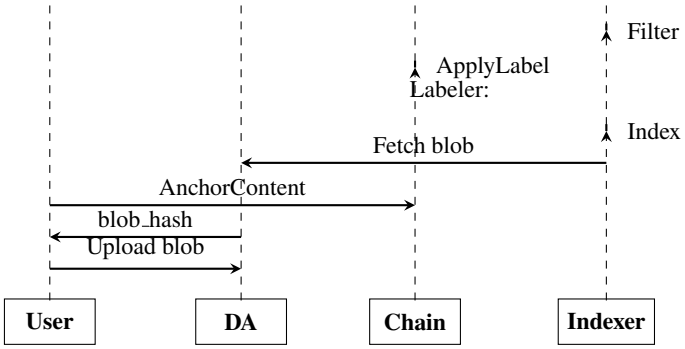


Figure 6: Post lifecycle: upload to DA, anchor on-chain, index, and label. Clients filter based on subscribed labelers.

### 6.6 Direct Messages

Direct messaging is a protocol-level primitive. Each account publishes a DM public key via `PublishDmKey`. Messages are encrypted end-to-end using X25519 key agreement and XChaCha20-Poly1305 (24-byte nonce), with only the ciphertext hash and DA reference recorded on-chain via `SendDm`. The domain for key derivation includes both sender and recipient identifiers, preventing key reuse across conversations. No plaintext ever touches the consensus layer.

### 6.7 Portability

A signed `PortableProfile` export packages a user's identity, key registry, credentials, graph roots, and content references into a verifiable bundle, enabling migration between indexers or DA backends without loss of provenance.

### 6.8 Standards Compatibility

ULI credentials are implemented as Poseidon commitments with Groth16 proofs (Section 7), which are more expressive than standard W3C Verifiable Credentials (VCs) [21]: they support range proofs, set membership, and anonymous attestation natively. However, enterprise integration requires compatibility with existing standards pipelines.

**DID resolution.** Each `did:uli:<id>` resolves to a DID Document containing the account's active public key, recovery methods, derived key grants, and service endpoints. The resolution is deterministic: any full node or indexer can serve as a DID resolver, eliminating the single-point-of-failure risk inherent in centrally hosted DID methods (e.g., `did:plc` in the AT Protocol).

**W3C VC bridge.** A bridge layer maps ULI ZK credentials to W3C Verifiable Presentation format. The Groth16 proof $\pi$ and public inputs $\vec{x}$ are embedded as a `proof` object in the VP JSON-LD envelope; the verifier checks $\pi$ against the on-chain verification key. This allows enterprise systems that consume OID4VP (OpenID for Verifiable Presentations) to accept ULI credentials without modification to their verification pipeline.

**Enterprise wallet compatibility.** ULI credentials can serve as the privacy-preserving backend for standards-compliant identity wallets (including wallets implementing the W3C VC / OID4VP stack): the wallet holds the user's ULI secret and generates Groth16 proofs on demand, while the presentation format conforms to W3C VC / OID4VP via the bridge layer. The bridge layer allows ULI to function as a credential backend without requiring changes to existing wallet presentation pipelines.

## 7. Zero-Knowledge Credentials

### 7.1 Design Philosophy

ULI applies zero-knowledge proofs *pointwise* on identity attributes, not on the consensus mechanism itself. The chain operates in the clear; ZK protects specific claims ("my attribute exceeds a threshold," "I am a unique human," "I authored this post") without revealing the underlying data. This avoids the throughput and complexity penalties of ZK-rollup architectures while providing meaningful privacy guarantees where they are most needed: at the identity and credential layer.

The credential lifecycle has three phases:

1. **Registration**: the user commits to a credential on-chain via `AddCredential`, publishing a Poseidon commitment $c$ and the verification key hash.
2. **Proving**: off-chain, the user generates a Groth16 proof using the SDK.
3. **Verification**: the proof is submitted on-chain; the consensus layer checks it and records the nullifier to prevent reuse.

Each credential has an optional expiration epoch. `expires_epoch = None` creates a permanent credential; `expires_epoch = Some(`$e$`)` creates a time-limited credential that must be re-attested after epoch $e$.

### 7.2 Proof System

The credential system uses Groth16 [4] on the BN254 curve. Groth16 was chosen for its constant-size proofs (3 group elements, approximately 128 bytes), fast verification (3 pairings), and mature tooling (arkworks, snarkjs, EVM-compatible verifiers). BN254 provides approximately 100-bit security against pairing-based attacks following the Kim–Barbulescu tower NFS improvement, lower than the 128-bit target of BLS12-381 used for consensus signatures. This is acceptable for credentials that can be re-issued (unlike irrevocable biometric commitments), and migration to BLS12-381 Groth16 or PLONK is supported by the circuit versioning mechanism (`RegisterCircuit`/`DeprecateCircuit`). The trusted setup is the principal drawback of Groth16: each circuit requires a separate Common Reference String (CRS) generated via MPC. The planned approach uses a shared Powers-of-Tau phase 1 ceremony (curve-generic, run once for BN254) followed by per-circuit phase 2 ceremonies (six total for the genesis circuits). This reduces operational cost compared to six fully independent ceremonies. Security requires at least one honest participant per ceremony. The choice of Groth16 over universal-setup systems

is a deliberate trade-off: constant-size proofs and ~3–5 ms verification (3 pairings) are critical when every validator must verify every proof within a 600 ms slot. PLONK proofs are 2–3× larger with 2–5× slower verification, directly reducing the per-block ZK throughput ceiling. The trusted setup is therefore a performance optimisation at genesis, not a permanent architectural commitment: circuit versioning via `RegisterCircuit` and `DeprecateCircuit` provides a migration path to transparent proof systems (PLONK, STARKs) that eliminate the trusted setup entirely; migration timing will be determined by circuit stability and prover performance benchmarks.

In-circuit hashing uses Poseidon [5] with parameters:

- Field: BN254 scalar field $\mathbb{F}_r$, $r \approx 2^{254}$.
- State size: 3 (rate 2, capacity 1).
- Rounds: 8 full + 57 partial ($R_F = 8$, $R_P = 57$).
- S-box: $x \mapsto x^5$.
- MDS matrix: Cauchy construction $M_{ij} = 1/(x_i + y_j)$.

This provides 128-bit security per the Poseidon specification [5].

### 7.3 Nullifier Scheme

Each proof produces a deterministic *nullifier* that prevents double-use without revealing the prover's identity.

Let $\chi$ be the chain identifier (`0x554C_495F_4153_4750`, ASCII "ULI_ASGP"), $\tau$ the circuit tag, $s$ the prover's secret, and $d$ a *structured domain separator*:

$$\nu = \text{Poseidon}(\chi, \tau, s, d), \tag{9}$$

$$c = \text{Poseidon}(\tau, s, b), \tag{10}$$

where $c$ is the commitment (with binding value $b$) stored on-chain at credential registration.

**Definition 4** (Domain separator)**.** *The domain separator $d$ is a Poseidon hash of a structured tuple:*

$$d = \textit{Poseidon}(\textit{scope},\ \textit{epoch\_window}), \tag{11}$$

*where scope is an application-defined identifier (app ID, service name, or relying-party public key) and epoch_window is an optional temporal qualifier. Two modes are supported:*

- ***Persistent*** *(epoch_window = 0): the nullifier is unique per credential per scope, preventing any re-use. Suitable for one-time verification (KYC onboarding, benefit claims).*
- ***Periodic*** *(epoch_window = e): the nullifier is unique per credential per scope* per epoch*, allowing re-attestation each epoch. Suitable for agent session re-attestation, periodic compliance checks, and any use case generating recurring verification volume.*

*The application specifies the mode in the verification request; the circuit enforces that $d$ is correctly constructed from the public inputs scope and epoch_window.*

The periodic mode is the mechanism that enables the 10-transaction-per-user-per-epoch workload assumed in the break-even analysis (Section 8.5): without a temporal component in $d$, each credential could produce only one nullifier per scope, and re-attestation (the primary volume source for agent-driven demand) would be impossible.

The consensus layer maintains a nullifier set and rejects proofs with previously recorded nullifiers. The chain identifier $\chi$ prevents cross-chain replay; the circuit tag $\tau$ prevents cross-circuit replay; the structured domain separator $d$ provides application-scoped and optionally time-scoped uniqueness.

**Nullifier set growth and pruning.** Periodic-mode nullifiers grow at $O(\text{verifications/epoch})$. At 2M verifications/epoch (break-even), each nullifier is 32 bytes, yielding ≈64 MB/epoch (≈8 GB/year) of consensus state. Persistent nullifiers cannot be pruned (they enforce one-time use indefinitely). Periodic nullifiers are semantically invalid after their epoch_window expires. The protocol prunes periodic nullifiers after $e + N_{\text{retain}}$ epochs (governance-set, default $N_{\text{retain}} = 2$), bounding steady-state storage to ≈128 MB regardless of history. Persistent nullifiers remain permanently; at KYC-frequency volumes, growth is ≈1 GB/decade.

### 7.4 Circuits

Six circuit types are defined, each with a unique tag $\tau \in \{1, \ldots, 6\}$ (Figure 7).

**1. Personhood ($\tau = 1$).** Proves the prover holds a registered personhood credential without revealing their identity. This enables Sybil resistance: the nullifier prevents using the same personhood credential twice in the same domain. Uniqueness of the credential depends on the issuing authority (external attestation service, biometric provider, or social graph scoring); the protocol provides the verification and nullifier infrastructure, not the uniqueness guarantee itself.

- Private: $s$ (secret), account_id.
- Public: $c$, $\nu$, $d$.
- Constraints: $c = \text{Poseidon}(1, s, \text{account\_id})$; $\nu = \text{Poseidon}(\chi, 1, s, d)$.

**2. AttributeRange ($\tau = 2$).** Proves an attribute $a \geq \theta$ without revealing the exact value (e.g., a credential attribute satisfies a lower bound).

- Private: $s$, $a$.
- Public: $c$, $\theta$, $\nu$, $d$.
- Constraints: $c = \text{Poseidon}(2, s, a)$; $a \geq \theta$; $\nu = \text{Poseidon}(\chi, 2, s, d)$.

**3. MerkleInclusion ($\tau = 3$).** Proves membership in a set of up to $2^{16}$ values via Poseidon-based Merkle proof. Leaf hashes: $h = \text{Poseidon}(\texttt{0x00}, v)$. Internal nodes: $h = \text{Poseidon}(\texttt{0x01}, h_l, h_r)$.

- Private: attribute value, Merkle path (16 siblings + bits).
- Public: Merkle root.
- Constraints: recomputed root matches public Merkle root.

**4. SelectiveDisclosure ($\tau = 4$).** Combines Merkle inclusion with a commitment: proves an attribute belongs to a disclosed set without revealing which element.

- Public: Merkle root, $c$, $\nu$, $d$.
- Constraints: Merkle inclusion holds; $c = \text{Poseidon}(4, s, a)$; $\nu = \text{Poseidon}(\chi, 4, s, d)$.

**5. AnonymousPost ($\tau = 5$).** Proves authorship without revealing the author. The content hash is a public input bound to the transaction's `content_hash` field at the consensus layer, not inside the circuit, preventing proof transplantation.

- Public: $c$, $\nu$, content_hash, $d$.
- Constraints: $c = \text{Poseidon}(5, s, \text{account\_id})$; $\nu = \text{Poseidon}(\chi, 5, s, d)$.

**6. EncryptedDM ($\tau = 6$).** Proves a direct message was correctly formed for a specific recipient without revealing the sender.

- Private: $s$, sender_id.
- Public: sender_commitment, $\nu$, recipient_id, ciphertext_hash, $d$.
- Constraints:
  sender_commitment = $\text{Poseidon}(6, s, \text{sender\_id})$;
  tag = $\text{Poseidon}(\text{recipient\_id}, \text{ct\_hash}, d)$;
  $\nu = \text{Poseidon}(\chi, 6, s, \text{tag})$.

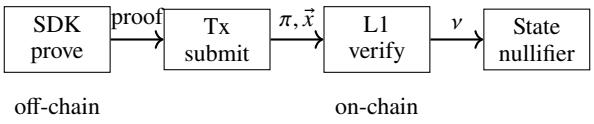


Figure 7: ZK credential flow. The SDK generates a Groth16 proof off-chain; the transaction carries proof $\pi$ and public inputs $\vec{x}$; the consensus layer verifies and records the nullifier $\nu$.

Figure 8 details the three-phase credential lifecycle including external issuance.

## 8. Economy

### 8.1 Capacity Staking

Users interact with ULI without holding tokens. *Providers* (typically application operators) stake tokens and receive capacity proportional to their stake:

$$\text{capacity}(s) = s \cdot C_{\text{rate}}, \tag{12}$$

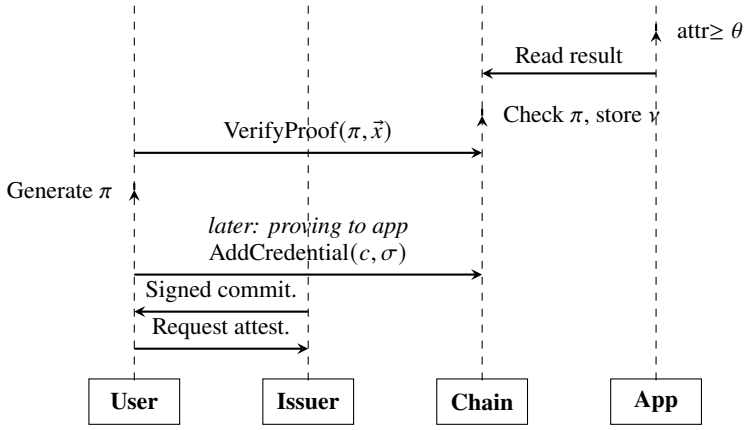


Figure 8: ZK credential flow: issuance by external issuer, on-chain registration, off-chain proof generation, and on-chain verification. The app learns only the proven claim (e.g., an attribute satisfies a threshold).

where $C_{\text{rate}} = 10$ capacity units per token (a governance parameter). Providers sponsor accounts via `SponsorAccount`, paying for on-chain operations from the provider's capacity balance. Capacity replenishes per epoch (non-cumulative): at each epoch boundary, the balance resets to capacity_per_epoch, not accumulated from prior epochs. The minimum provider stake is 1,000 tokens.

**Definition 5** (Transaction Admission)**.** *Let $P$ be a provider with stake $s_P$. The provider maintains two balances:*

1. ***Rate limit*** *(anti-spam): per-epoch transaction quota $q_P = s_P \cdot C_{rate}$, reset at each epoch boundary.*
2. ***CC balance*** *(payment): Capacity Credits minted by burning ULI at oracle price (Definition 6).*

*Each operation $o$ has a CC cost $w(o)$: 1 CC for standard operations, 10 CC for verifications. A sponsored transaction succeeds iff $q_P > 0$* and *$CC_P \geq w(o)$, after which $q_P \leftarrow q_P - 1$ and $CC_P \leftarrow CC_P - w(o)$. ULI is permanently destroyed at CC minting time; the aggregate epoch burn is given by Equation 14.*

Admission is enforced not only at execution but also at mempool insertion and block proposal, preventing unsponsored transactions from occupying block capacity (see Section 12, DoS resistance).

This model eliminates the cold-start problem of token-gated networks. A social application can onboard millions of users by operating as a provider; individual users need never purchase, hold, or even be aware of the underlying token. The cost of user acquisition is borne by application operators, who interact with the token on behalf of their users, so end users never need to navigate cryptocurrency exchanges.

The replenishment-not-accumulation policy ensures that unused capacity does not roll over, maintaining predictable per-epoch resource consumption and preventing providers from hoarding capacity for burst attacks. The capacity model provides a natural rate limit: providers cannot exceed their per-epoch allocation regardless of demand.

**Anti-spam limitations.** The minimum provider stake of 1,000 tokens yields $1{,}000 \times 10 = 10{,}000$ capacity units per epoch, permitting up to 10,000 transactions every ≈3 days. At low token prices this may be insufficient to deter spam. The governance-adjustable parameters ($C_{\text{rate}}$ and minimum stake) allow the protocol to increase the cost of spam as the network matures, but the initial configuration prioritises low barriers to entry over aggressive anti-spam guarantees. Applications may supplement protocol-level rate limiting with additional application-layer controls.

**Provider exit lifecycle.** Provider exit follows a three-phase lifecycle: (1) *Deactivation*: the provider submits `DeactivateProvider`, which marks the provider as inactive and prevents new sponsorships; (2) *Grace period*: 7 epochs (≈21 days) during which sponsored users can migrate to another provider; the provider's stake remains locked; (3) *Unbonding*: after the grace period plus the standard 14-epoch unbonding period, the provider reclaims stake via `ClaimProviderUnbonded`. Re-deactivation of an already deactivated provider is rejected.

### 8.2 Validator Staking

Validators register with a self-stake, BLS12-381 public key, and commission rate (in basis points, ≤ 10,000). Delegators may stake to any active validator via `DelegateStake`. A validator's voting power equals self-stake plus delegated stake.

Undelegation initiates an unbonding period. Up to 32 independent unbonding entries may be active simultaneously, each with its own maturity epoch. Tokens become claimable strictly after the completion epoch.

Validators may be jailed for equivocation (Section 4). A jailed validator's stake continues to be slashable during the jail period but does not earn rewards. After the jail period expires (minimum 7 epochs, approximately 21 days), the validator must submit `Unjail` to resume participation. Delegators to a slashed validator lose a proportional share of their delegation, creating incentive to diversify delegations across validators and monitor their behaviour.

### 8.3 Schema Deposits

Schema registration requires an anti-spam deposit. The deposit is not a governance gate: any account may register a schema in any namespace by paying the deposit. The deposit remains locked while the schema is active, creating an economic cost to namespace pollution while preserving permissionless innovation.

### 8.4 Token Supply and Emission Schedule

The native token (ULI) has a fixed supply of $10^{10}$ (10 billion). Genesis distribution and vesting schedules are specified in the whitepaper [22], Section 7. Staking rewards are funded from a pre-allocated emission reserve (3.5B ULI, 35% of supply), not from inflation.

The per-epoch reward declines geometrically:

$$R(e) = R_0 \cdot \lambda^{\lfloor e/H \rfloor}, \tag{13}$$

where $R_0 = 1{,}000{,}000$ ULI is the initial per-epoch reward, $\lambda = 0.5$ (halving), and $H = 488$ epochs ($\approx 4$ years at $\approx 122$ epochs/year).

### 8.5 Fee Burn

Staking alone does not create sustained token demand: a provider stakes once and uses the rate limit indefinitely. The CC mechanism closes this gap: providers must continuously burn ULI to mint CCs, and CCs are consumed by every transaction.

The aggregate ULI burned by provider $P$ in epoch $e$ is the total ULI converted to CCs during that epoch:

$$B_P(e) = \sum_{\text{mints in } e} \frac{\text{CC}_{\text{minted}} \cdot P_{\text{CC}}}{P_{\text{ULI}}(e)}. \tag{14}$$

Burned tokens are permanently removed from supply ($S_{\max}$ decreases). They are not sent to the treasury or redistributed. This creates a deflationary force that grows with network activity: more usage → more CC consumption → more ULI burned at minting → buy pressure.

**Unit economics and target workload.** Transaction costs are denominated in CCs (Definition 6): standard operations (account creation, key rotation, content anchoring) consume 1 CC ($0.001); verification transactions (`VerifyCredentialProof`) consume 10 CC ($0.01) each, reflecting their higher computational cost and economic value. The ULI burned per transaction depends on the oracle price: at $P_{\text{ULI}} = \$0.01$, a verification burns 1.0 ULI; at $P_{\text{ULI}} = \$0.10$, the same verification burns 0.1 ULI. The AnchorGraph transaction type supports batching multiple content anchors into a single 1-CC transaction, reducing per-item cost.

**Capacity Credits.** A Capacity Credit (CC) is a non-transferable, non-tradeable prepaid usage unit with a fixed fiat peg:

**Definition 6** (Capacity Credit conversion)**.** *Given the oracle-reported ULI price $P_{ULI}$ and the governance-set CC peg $P_{CC} = \$0.001$, the number of CCs minted from a burn of $b$ ULI is:*

$$CC_{minted}(b) = b \times \frac{P_{ULI}}{P_{CC}}. \tag{15}$$

The ULI burned per transaction is $\text{CC}_{\text{cost}} \times P_{\text{CC}}/P_{\text{ULI}}$ (see unit economics above for per-operation CC costs).

The oracle computes a *stake-weighted median* at each epoch boundary. Active, non-jailed validators submit `UpdateOraclePrice` transactions during the epoch. At the first slot of the new epoch, `finalize_oracle_price()` sorts submissions by price and selects the value at the 50%-of-cumulative-stake crossing point. Quorum: ≥3 submitting validators. Deviation cap: submissions >20% (`MAX_DEVIATION_BPS` = 2000) from the current price

are discarded. If no valid quorum is reached, the last valid price carries forward (idempotency guard via `LAST_ORACLE_EPOCH_KEY`).

Capacity Credit minting uses a TWAP (time-weighted average price) computed over a rolling window of the last 7 finalized epoch prices (`TWAP_WINDOW = 7`). The TWAP ring buffer records each epoch's median price; the CC mint price is the median of the stored prices. This smooths short-term volatility and limits single-epoch oracle manipulation.

*Bootstrap mode.* Before exchange listing, the Foundation sets $P_{\text{ULI}}$ via governance (trusted oracle), transitioning to the validator-driven median when the active set is sufficiently decentralised. A corrupted validator quorum can mis-price CCs for at most one epoch; the deviation cap and TWAP smoothing bound the damage.

Token demand mechanisms and provider unit economics (including fiat-pegged CC cost tables) are detailed in the whitepaper [22], Section 7.

### 8.6 Reward Schedule and Validator APR

Each epoch distributes $R(e)$ ULI (Equation 13) among active validators proportional to their effective stake. Let $s_v$ be a validator's total stake (self + delegated), $S$ the aggregate stake, and $\rho_v$ the commission rate in basis points. The validator's gross reward per epoch is:

$$r_v = R(e) \cdot \frac{s_v}{S}. \tag{16}$$

The validator retains commission $\rho_v/10{,}000 \cdot r_v$; delegators share the remainder proportional to their delegation.

With one epoch $\approx$ 3 days ($\approx$ 122 epochs/year), the naïve annualised reward rate would be $R_0 \cdot 122/S_{\text{total}}$. However, distributing the full $R(e)$ when staking participation is low produces disproportionately high APR (e.g. 24.4% at 5% staked), enabling emission farming by early participants.

**Definition 7** (Effective emission)**.** *The effective emission per epoch is:*

$$R_{eff}(e) = R(e) \cdot \min\left(1,\ \frac{S_{total}}{S_{target}}\right), \qquad S_{target} = 0.25 \times S_{\max} = 2.5B\ ULI. \tag{17}$$

When $S_{\text{total}} < S_{\text{target}}$, emission scales down linearly with participation. Unissued rewards ($R(e) - R_{\text{eff}}(e)$) remain in the reserve pool, extending the emission schedule beyond the nominal schedule. No new tokens are minted; rewards are drawn from the pre-allocated reserve, but they increase circulating supply, exerting sell-pressure indistinguishable from inflation until the market absorbs it.

The annualised reward rate is:

$$\text{APR} = \frac{R_{\text{eff}}(e) \cdot 122}{S_{\text{total}}}. \tag{18}$$

| Staked (% of supply) | $S_{\text{total}}$ | $R_{\text{eff}}$/epoch | Year-1 APR |
|---:|---:|---:|---:|
| 5% | 500M | 200,000 | 4.9% |
| 10% | 1.0B | 400,000 | 4.9% |
| 25% | 2.5B | 1,000,000 | 4.9% |
| 50% | 5.0B | 1,000,000 | 2.4% |

Table 3: Year-1 APR under emission scaling ($R_0$ = 1,000,000, $S_{\text{target}}$ = 2.5B). APR is capped at $\approx$ 4.9% regardless of how low staking participation falls. The cap operates by reducing absolute emission, not by redistributing it; unissued tokens remain in reserve.

After each halving ($H$ = 488 epochs $\approx$ 4 years), $R_0$ halves and APR adjusts accordingly. At 25% staked and post-first-halving, APR $\approx$ 2.4%. Combined with $\alpha$-derived fee revenue from genesis, total validator yield exceeds the headline APR.

### 8.7 Slashing Economics

Equivocation slashing removes 10% of the offending validator's stake (Equation 4). Slashed tokens are permanently burned, not redistributed, preventing perverse incentives where validators benefit from competitors' slashing. The validator is jailed for a minimum of 7 epochs ($\approx$ 21 days), during which no rewards accrue.

Delegators to a slashed validator lose a proportional share of their delegation, creating a strong incentive to:

1. Diversify delegations across multiple validators.
2. Monitor validator uptime and behaviour.
3. Prefer validators with proven operational track records.

### 8.8 Security Budget and Terminal Economics

The minimum cost to compromise consensus safety is the cost of acquiring $q = \lfloor 2S/3 \rfloor + 1$ stake. Table 4 provides lower-bound estimates at three staking levels and three token prices. These assume spot-price acquisition with no market impact; in practice, acquiring a supermajority would exhaust available float and drive price up by orders of magnitude.

| Staked | $S_{\text{total}}$ | >2S/3 req. | Min. acquisition cost (USD) | | |
|---|---|---|---|---|---|
| | | | \$0.01 | \$0.10 | \$1.00 |
| 5% | 500M | 334M | \$3.3M | \$33M | \$334M |
| 25% | 2.5B | 1.67B | \$16.7M | \$167M | \$1.67B |
| 50% | 5.0B | 3.34B | \$33.4M | \$334M | \$3.34B |

Table 4: Lower-bound cost of consensus attack ($> 2S/3$ stake at spot price, ignoring market impact, slippage, and slashing).

An attacker who attempts equivocation faces a 10% slash per detected offence, so the expected loss from a failed attack is:

$$L_{\text{attack}} = 0.1 \cdot s_{\text{attacker}} \cdot p_{\text{detect}}, \tag{19}$$

where $p_{\text{detect}}$ is the detection probability (approaching 1 for on-chain equivocation, since any honest validator can submit evidence). For a rational adversary, the attack is profitable only when the expected gain exceeds $L_{\text{attack}}$ plus the opportunity cost of forgone staking rewards over the 7-epoch jail period.

**Early-network security.** At genesis, the foundation stakes its entire allocation (1.5B ULI, 15% of supply) to validators, establishing a minimum staking floor independent of market participation. Combined with team tokens (15%, vesting, not tradeable), the emission reserve (35%, locked), and ecosystem & grants allocation (20%, governance-gated, capped at 0.5% of supply per quarter = 50M ULI/quarter), the effective free float is ≤15% of supply in year one (early supporters 10% + initial liquidity 5%), rising to at most 17% by year-end if grants are fully disbursed. All locks and disbursement caps are consensus-enforced, not contractual: an attacker cannot acquire tokens that do not circulate. During the permissioned phase, the Foundation controls sufficient stake to pass governance votes unilaterally; grant disbursement guards are reputational self-constraints until the permissionless transition, not cryptoeconomic guarantees. The initial validator set is permissioned (progressive decentralisation). Year 1–3 security is funded by foundation stake and vesting locks, not by market demand, a bootstrap condition shared by all new PoS chains but not disguised here by analogy with mature networks. Market-driven security activates when $S_{\text{total}} > S_{\text{target}}$. Progressive decentralisation follows three verifiable milestones: (1) at 10% staked, the validator set opens to any operator posting a bond $\geq B_{\min}$ and demonstrating state synchronisation; (2) at $S_{\text{target}}$ (25%), full permissionless entry; (3) at epoch $E_{\text{fallback}}$ (governance-set, default 366 ≈ 3 years), permissionless entry activates unconditionally regardless of staking level.

**Pre-adoption token demand.** The token valuation thesis is usage-based: pre-adoption holders face dilution from emission with no offsetting burn, compensated through staking rewards (APR capped at ≈4.9%). The foundation stake floor ensures consensus safety regardless of market price; see the whitepaper [22], Section 7 for the full demand analysis.

**Break-even activity level.** With burn-split $\alpha = 0.5$, the token is deflationary when $B_{\text{total}} \geq 2 \times R_{\text{eff}}$. At year-1 parameters (25% staked, $R_{\text{eff}}$ = 1M/epoch), this requires 2M verifications/epoch (≈7.7 sustained TPS at $P_{\text{ULI}}$ = \$0.01). Break-even TPS scales linearly with price: $\text{TPS}_{\text{BE}} = 7.7 \times P_{\text{ULI}}/\$0.01$. Table 5 translates this to annual volumes.

Table 5: Break-even annual verifications at $P_{\text{ULI}}$ = \$0.01. Global identity verification volume: ≈75B checks/year.

| Staked | Break-even | Annual tx | Users (2/yr) | Users (5/yr) |
|---|---|---|---|---|
| 5% | 1.5 TPS | 48.6M | 24.3M | 9.7M |
| 10% | 3.1 TPS | 97.4M | 48.7M | 19.5M |
| 25% | 7.7 TPS | 243M | 122M | 48.7M |

At 5% staking, break-even is ≈49M verifications/year (0.06% of global volume). Provider unit economics, cold-start scenarios, and the stabilising feedback loop (low price → more ULI burned per CC → stronger deflation) are detailed in the whitepaper [22], Section 7.

**Cold-start path and sensitivity.** Emission scaling reduces break-even proportionally to staking: 1.5 TPS at 5% staked, 4.6 TPS at 15%, 7.7 TPS at 25%. Figure 9 plots both the break-even TPS and the token price required for a $100M attack cost; the safe zone is above both curves. See the whitepaper [22], Section 7 for the full cold-start analysis.

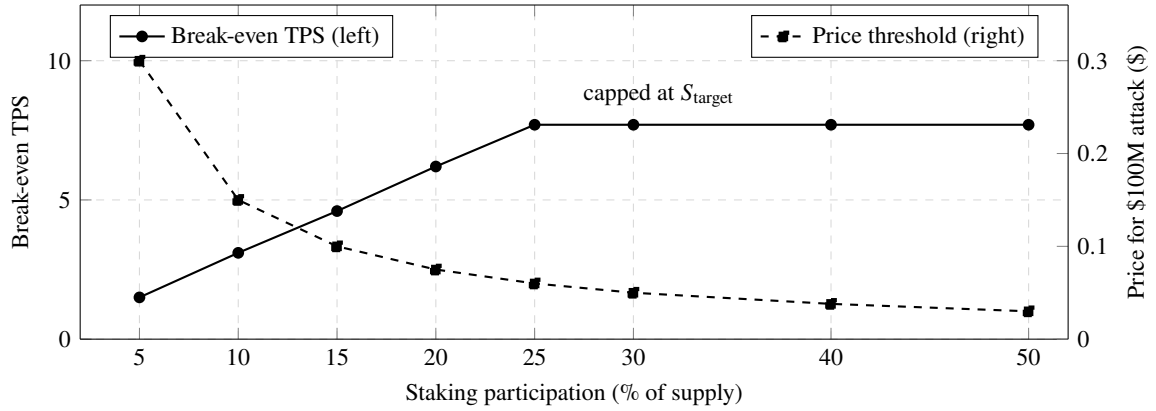


Figure 9: Security and inflation thresholds vs. staking participation (year 1). Solid (left axis): sustained identity-verification TPS for burn to equal emission; above the curve the token is deflationary. Dashed (right axis): token price at which a $> 2S/3$ attack costs $100M (spot, ignoring slippage). The safe zone is the region above both curves.

**Terminal state.** A burn-split parameter $\alpha \in [0, 1]$ ($\alpha = 0.5$ from genesis) routes fraction $\alpha$ of each epoch's CC burn to validators as fee revenue; fraction $(1 - \alpha)$ is permanently destroyed. Validators earn $R_{\text{eff}} + \alpha \times B_{\text{total}}$ per epoch from genesis. As emission halves (Equation 13), $\alpha \times B_{\text{total}}$ becomes the dominant validator income. The emission reserve (3.5B ULI) exceeds total uncapped emission (976M); there is no reserve-exhaustion cliff. Fee burn ensures continuous token demand as long as the network has usage; see the whitepaper [22], Section 7 for the full terminal-state analysis.

## 9. Moderation

The moderation architecture (labeler system, client-side filtering, on-chain appeals, and the structural argument against opaque ranking) is described in the whitepaper [22], Section 8. This section specifies the formal properties relevant to indexer implementation.

**Labeler parameters.** Deposit: 10,000 ULI. Label types: open strings, max 64 characters. Subscriptions per user: max 64. Labels may be negated (retracted) by the issuing labeler. Appeals are on-chain, timestamped, and do not automatically remove labels.

**Bridging-based ranking.** The ASGP graph structure makes bridging metrics [15] computable at the indexer layer: cross-cluster connectivity is measurable on the ZK graph root, enabling ranking that promotes content with cross-partisan approval (structurally similar to the Community Notes model) without embedding an editorial policy in consensus. Bail et al. [14] showed that naive exposure to opposing views can *increase* polarisation; bridging-based ranking avoids this by surfacing content that receives approval *across* clusters, not content that merely contradicts the user's existing views.

This architecture does not make manipulative ranking impossible. It makes it *detectable and switchable*.

## 10. Algorithms and Complexity

This section formalises four core algorithms whose informal descriptions appear in earlier sections, and provides worst-case complexity bounds. The bridging coefficient is defined first, as it is used by the subsequent trust score algorithm.

**Algorithm 1:** Compute Bridging Coefficients

**Input:** Graph $G = (V, E)$, partition $\{P_1, \ldots, P_c\}$
**Output:** Bridging coefficients $\beta[v]$ for all $v \in V$
**foreach** $v \in V$ **do**
  cross $\leftarrow 0$;
  **foreach** $(v, u) \in E$ **do**
    **if** *partition*$(u) \neq$ *partition*$(v)$ **then**
      cross $\leftarrow$ cross + 1;
  $\beta[v] \leftarrow \text{cross}/\deg(v)$;
**return** $\beta$

### 10.1 Bridging Metric for Indexer Ranking

**Definition 8** (Bridging Coefficient)**.** *Let $\{P_1, \ldots, P_c\}$ be a partition of $V$ into $c$ communities (computed via Louvain or spectral clustering). For vertex $v \in P_i$, the bridging coefficient is:*

$$\beta(v) = 1 - \frac{\left|\{(v, u) \in E : u \in P_i\}\right|}{\deg(v)}, \tag{20}$$

*i.e., the fraction of $v$'s edges that cross community boundaries. $\beta(v) = 0$ if all neighbours are in the same community; $\beta(v) \to 1$ if $v$ connects predominantly to other communities.*

**Complexity.** $O(|V| + |E|)$ given a precomputed partition. The partition itself (Louvain) runs in $O(|E|)$ expected time. Bridging coefficients are precomputed by indexers at epoch boundaries and cached for query-time ranking.

**Partition stability.** Louvain community detection is non-deterministic: different random seeds or tie-breaking orders can produce different partitions, which in turn alter bridging coefficients and, through Equation 21, trust scores. To mitigate this, indexers should either fix the random seed per epoch (ensuring reproducibility within an epoch) or use ensemble methods (averaging $\beta$ over multiple Louvain runs). The protocol does not mandate a specific partition algorithm; indexers are free to use spectral clustering, label propagation, or any community detection method, and users choose which indexer's scores to trust. This variability is a deliberate consequence of delegating graph analysis to the indexer layer rather than ossifying a single algorithm in consensus.

### 10.2 Graph Trust Score

The trust score $\tau(v)$ quantifies how well a vertex $v$ is embedded in the honest core of the vouching subgraph. The computation is a weighted PageRank variant with a cross-cluster bridging bonus, using the bridging coefficients $\beta$ from Definition 8.

**Definition 9** (Trust Score)**.** *Let $G_{\mathcal{V}} = (V, E_{\mathcal{V}}, w)$ be the weighted vouching subgraph where $w : E_{\mathcal{V}} \to [0, 1]$ assigns vouch weights. The trust score $\tau : V \to \mathbb{R}_{\geq 0}$ is the fixed point of the iterative update (not a probability distribution, since the bridging coefficients $\beta(u)$ cause mass leakage when $\beta < 1$):*

$$\tau^{(k+1)}(v) = \gamma \sum_{u \in N^-(v)} \frac{w(u, v) \cdot \beta(u)}{\sum_z w(u, z)} \tau^{(k)}(u) + \frac{1 - \gamma}{|V|}, \tag{21}$$

*where $\gamma \in (0, 1)$ is the damping factor, $N^-(v)$ is the in-neighbour set, and $N^+(u)$ is the out-neighbour set.*

**Complexity.** Each iteration processes every edge once: $O(|E_{\mathcal{V}}|)$. With $k_{\max}$ iterations, total cost is $O(|E_{\mathcal{V}}| \cdot k_{\max})$. In practice, $k_{\max} \leq 20$ suffices for convergence on social graphs. The algorithm is executed by indexers at epoch boundaries, not by validators.

### 10.3 Sybil Resistance via Social Vouching

Here $d(u, v)$ is the shortest-path distance in the full graph (used as a proximity decay) and $M$ caps the number of vouches considered to bound computation.

**Algorithm 2:** Compute Graph Trust Scores

**Input:** Vouching graph $G_{\mathcal{V}} = (V, E_{\mathcal{V}}, w)$, damping $\gamma$, iterations $k_{\max}$, bridging coefficients $\beta$
**Output:** Trust scores $\tau[v]$ for all $v \in V$
**foreach** $v \in V$ **do**
  $\tau[v] \leftarrow 1/|V|$
**for** $k \leftarrow 1$ **to** $k_{\max}$ **do**
  **foreach** $v \in V$ **do**
    $s \leftarrow 0$;
    **foreach** $u \in N^-(v)$ **do**
      $s \leftarrow s + \dfrac{w(u,v) \cdot \beta[u]}{\sum_{z \in N^+(u)} w(u,z)} \cdot \tau[u]$;
    $\tau'[v] \leftarrow \gamma \cdot s + (1-\gamma)/|V|$;
  $\tau \leftarrow \tau'$;
**return** $\tau$

**Algorithm 3:** Vouch Score with Sybil Threshold

**Input:** Account $v$, vouching graph $G_{\mathcal{V}}$, threshold $\theta$, decay $\delta \in (0,1)$, max vouches $M$
**Output:** Boolean: $v$ passes Sybil check
$V_v \leftarrow$ set of accounts that vouch for $v$;
$V_v \leftarrow V_v$ restricted to $|V_v| \le M$ highest-$\tau$ vouchers;
$\sigma \leftarrow 0$;
**foreach** $u \in V_v$ **do**
  $\sigma \leftarrow \sigma + w(u,v) \cdot \tau(u) \cdot \delta^{d(u,v)}$;
**return** $\sigma \ge \theta$

**Observation (Sybil Bound).** Under honest-majority vouching (> 50% of vouchers are honest), a Sybil cluster $C$ of $m$ colluding accounts satisfies $\sum_{v \in C} \sigma(v) \le m \cdot \tau_{\max} \cdot M$, where $\tau_{\max}$ is the maximum trust score of any colluding voucher. If honest accounts have trust scores $\ge \theta/M$ and colluding accounts have $\tau < \theta/M$, then no Sybil account in $C$ passes the threshold.

*Caveat*: This bound assumes that the trust-score algorithm assigns low $\tau$ values to colluding accounts, which depends on the graph topology and the stability of the community partition (Section 10.1). The bound is a necessary condition, not a sufficient guarantee of Sybil resistance; small-scale collusion attacks where attackers establish genuine-looking connections to the honest core remain difficult to detect through topology alone [11]. For use cases where probabilistic Sybil resistance is insufficient, the protocol provides credential-based uniqueness: an external issuer certifies the credential, and the global nullifier set prevents double-use. The social graph and external attestation are complementary tiers, not substitutes.

**Complexity.** Per account: $O(M)$ for the vouch score, plus $O(|V| + |E|)$ amortised for shortest-path distances (precomputed via BFS at epoch boundaries).

### 10.4 Capacity Metering

**Complexity.** $O(1)$ per transaction: two comparisons and two decrements. Epoch reset is $O(1)$ (triggered lazily on first transaction of a new epoch); CC balance is replenished by explicit mint transactions (Definition 6), not by epoch reset. This is the only algorithm in this section executed by validators during consensus; the others run on indexers.

## 11. Performance Bounds

This section derives throughput, storage, and cost bounds from the protocol parameters defined in Table 2 and the codebase constants, supplemented by single-node empirical benchmarks. Geo-distributed multi-validator measurements will follow testnet deployment.

**Algorithm 4:** Capacity Metering (dual-balance)

**Input:** Transaction tx, provider $P$, current epoch $e$
**Output:** Accept or reject
**if** $e > P.last\_epoch$ **then**
  $P.q \leftarrow P.\text{stake} \cdot C_{\text{rate}}$ ; `// rate-limit reset`
  $P.\text{last\_epoch} \leftarrow e$;
$w \leftarrow \text{cc\_cost(tx)}$ ; `// 1 CC standard, 10 CC verify`
**if** $P.q > 0$ ***and*** $P.cc \geq w$ **then**
  $P.q \leftarrow P.q - 1$;
  $P.\text{cc} \leftarrow P.\text{cc} - w$;
  **return** *accept*
**else**
  **return** *reject*

### 11.1 Throughput Analysis

With MAX_TXS_PER_BLOCK = 256 and SLOT_DURATION_MS = 600, the analytical maximum throughput is:

$$\text{TPS}_{\text{max}} = \frac{256}{0.6} \approx 426.7 \text{ transactions/second.} \tag{22}$$

**Empirical benchmark (single-node, dev mode).** Configuration: single validator on AMD Ryzen 5 5600X (6C/12T), 32 GB DDR4-3200, NVMe SSD, Linux 6.18; RocksDB state store; in-memory DA backend; loopback consensus exercising the full propose–vote–QC–commit path with active BLS signing; no network round-trip. A load-testing harness with 32 concurrent WebSocket connections submitted `CreateAccount` transactions (each with a fresh keypair, generated outside the latency timer) at 1 s block intervals (256 tx/block cap) for 30 s.

Table 6: Single-node load test results. Committed TPS and block interval are computed over the block window (first to last observed block), not test duration.

| Metric | Value |
|---|---|
| Submit TPS (offered load) | 943 |
| Committed TPS (block window) | 253 |
| Avg. block interval | 1,012 ms |
| Avg. block fill | 248/256 (97%) |
| Submit RTT avg. (RPC accept) | 4.8 ms |
| Rejected transactions | 0 |

The workload exercises account-state writes, Sr25519 signature verification, and incremental SMT state-root computation (the sponsorship-exempt onboarding path); it does not exercise per-account nonce sequencing, capacity metering, or the DA and ZK verification paths, which require provider setup and are planned for the representative-mix testnet benchmarks. At 97% block fill against the 256-transaction cap, throughput is bounded by block capacity rather than execution: committed TPS constitutes an execution-layer upper bound. Protocol throughput under BFT with geographically distributed validators is bounded additionally by network round-trip and QC aggregation latency and will be measured on the public testnet.

**BFT overhead.** Sustained throughput under BFT with $n$ validators depends on the QC aggregation latency: the leader must collect $> 2S/3$ stake in BLS signatures before proposing the next block. BLS12-381 aggregate verification amortises to $O(1)$ pairings per QC (a single multi-pairing), so verification cost is dominated by network round-trip time rather than cryptographic computation. The 600 ms slot target assumes dedicated validator hardware with low-latency interconnects; sustained throughput with geographically distributed validators will be measured during testnet deployment.

Table 7 summarises representative transaction types, their approximate serialised sizes, and capacity costs.

Table 7: Representative transaction costs.

| Transaction type | Size (bytes) | Capacity |
|---|---|---|
| CreateAccount | ~128* | 1 |
| RotateKey | ~160* | 1 |
| AnchorContent | ~256* | 1 |
| AnchorGraph | ~192* | 1 |
| VerifyCredentialProof | ~512* | 10 |
| RegisterSchema | variable | deposit |
| DelegateStake | ~96* | 1 |
| SendDm | ~320* | 1 |

*Estimated from Borsh serialisation of struct fields.

### 11.2 State Size Analysis

Consensus state is persisted across 25 RocksDB column families, of which 21 are included in the state root computation. Four column families (metadata, sync state, oracle cache, and temporary indices) are excluded as they contain node-local data.

Table 8: Major state column families.

| Column family | Data stored | Per-account |
|---|---|---|
| Accounts | Full account state | ~2 KB† |
| Keys | Key → ID index | ~64 B† |
| Validators | Staking info | ~512 B† |
| Schemas | Schema definitions | ~1 KB† |
| Nullifiers | ZK nullifier set | 32 B each |
| Labels | Label records | ~128 B† |
| Credentials | Credential commitments | ~96 B† |

†Estimated from struct field sizes; not benchmarked.

State root computation uses an incremental sparse Merkle tree (SMT), reducing per-block state root cost to $O(m \log n)$ for $m$ modified keys out of $n$ total entries.

The dual Merkle root design uses Blake3 for the integrity root (fast native computation) and Poseidon for the ZK root (efficient in Groth16 circuits).

### 11.3 Data Availability Size

Each gossip message is bounded by MAX_TRANSMIT_SIZE = 1 MiB. Representative blob sizes:

- Content anchor: ~256 bytes (hash + metadata).
- Graph edge: ~128 bytes (source, target, type, encrypted payload).
- Encrypted DM: variable, typically 1–4 KB (ciphertext + AEAD tag + nonce).

Erasure coding introduces an $N/K$ expansion factor. The default expansion factor is $N/K = 2$, where $K$ scales dynamically with blob size ($K = \max(4, \lceil \text{len}/512 \rceil)$) and the expansion factor is configurable via `ErasureConfig` (range 2–8). For minimum-size blobs this yields $K = 4, N = 8$. Storage overhead is $N/K\times$, allowing latency-sensitive blobs (DMs) to use lower redundancy than archival content.

### 11.4 Validator Computational Load

Per-slot, a validator performs:

1. Verify up to 256 transactions (signature checks, nonce validation, state preconditions).
2. Execute state transitions and compute the state root.
3. Aggregate BLS12-381 votes into a QC.
4. Validate DA references (retrieve and verify erasure-coded blobs before voting).

BLS12-381 aggregate verification uses a single multi-pairing for $n$ signers, amortising the cost to approximately 2 pairings regardless of validator count. Individual Sr25519 signature verification runs at ~10 µs per signature on modern hardware, so 256 transactions require ~2.56 ms of signature verification, well within the 600 ms slot window.

### 11.5 ZK Proof Generation

The proof system uses Groth16 on BN254 with MAX_PROOF_SIZE = 4,096 bytes (accommodating compressed proofs with public inputs). A Groth16 proof consists of 3 group elements (~128 bytes), but the transaction also carries public inputs and metadata.

All six circuits are fully implemented as R1CS constraint systems using the arkworks library (ark-bn254 + ark-groth16). Approximate constraint counts:

- Personhood: ~10K constraints (2 Poseidon hashes).
- AttributeRange: ~12K constraints (2 hashes + range check).
- MerkleInclusion: ~50K constraints (16-level Poseidon tree).
- SelectiveDisclosure: ~55K constraints (Merkle + commitment).
- AnonymousPost: ~10K constraints (2 hashes).
- EncryptedDM: ~15K constraints (3 hashes).

Typical Groth16 provers achieve ~100K constraints/second on a single core [4], yielding 0.1–0.6 s per proof for these circuits. Verification is constant-time: 3 pairings ≈ 3–5 ms.

**ZK verification throughput ceiling.** A block containing only `VerifyCredentialProof` transactions would require 256 × 3–5 ms = 768–1,280 ms of sequential pairing computation, exceeding the 600 ms slot. In practice, three factors relax this bound: (1) the realistic transaction mix is dominated by non-ZK operations (content anchors, graph updates, DMs); a realistic production mix is expected to contain ~5% ZK transactions; (2) BN254 pairing verification parallelises across cores (a 4-core validator processes 4 proofs concurrently, reducing wall-clock time to ~192–320 ms for 256 proofs); (3) multi-pairing batching amortises the final exponentiation across proofs in the same block. At 5% ZK mix, a full block contains ~13 proofs (~39–65 ms), well within the slot. To prevent ZK verification from dominating block processing time, the protocol enforces `MAX_ZK_TXS_PER_BLOCK` = 32. This limit is checked in the mempool (batch construction), consensus (block validation), and runtime (execution). At 32 ZK proofs per block with 4-core parallelism, worst-case verification time is ~24–40 ms, well within the 600 ms slot. Beyond parameter tuning (block size, slot duration), the primary scaling mechanism is *recursive proof aggregation*: an off-chain aggregator combines $N$ ZK proofs into a single recursive proof verified once per block. Efficient recursion over Groth16/BN254 requires non-native field arithmetic (cycle-of-curves or folding schemes such as Nova); this is a primary motivation for the PLONK/STARK migration path described in Section 7.2. Recursive aggregation is expected to precede rather than follow proof-system migration, as the scaling bottleneck manifests before circuit stability concerns justify the migration cost. The circuit versioning mechanism (`RegisterCircuit`/`DeprecateCircuit`) supports this transition without chain halt.

### 11.6 Protocol Comparison

Table 9 compares operational costs across decentralised social protocols.

Table 9: Operation cost comparison across protocols.

| Operation | ULI | AT Proto | Farcaster | DeSo | Nostr |
|---|---|---|---|---|---|
| Account | 1 cap. | Free | Hub fee | DeSo tx | Free |
| Post | 1 cap. | Free | Hub stor. | DeSo tx | Relay |
| Recovery | On-chain | PLC log | ETH-based | None | None |
| ZK cred. | On-chain | N/A | N/A | N/A | N/A |
| Encryption | Per-edge | None | None | None | NIP-44 |
| Agent auth. | Derived key | App pwd | N/A | N/A | N/A |

Among the protocols surveyed in Section 13, ULI provides protocol-level key recovery, zero-knowledge credentials, encrypted social graph, and agent attestation simultaneously. AT Protocol (Bluesky) offers portable identity and composable moderation without a blockchain or token, but lacks ZK credentials, graph privacy, and sovereign recovery (its `did:plc` method relies on a centrally hosted rotation log). ULI provides the cryptographic substrate that AT Protocol's architecture does not address.

## 12. Security Analysis

### 12.1 Consensus Safety and Liveness

Safety follows from Theorem 1: the 2-chain commit rule, combined with monotonic `last_voted_view` and the quorum intersection lemma, prevents conflicting commits under the assumption $n \geq 3f + 1$. Liveness is guaranteed under partial synchrony: after GST, an honest leader will produce a block with a valid QC within bounded time. Equivocation is deterred by a 10% stake slash and 7-epoch jail.

### 12.2 ZK Soundness

Groth16 provides computational soundness under the generic bilinear group model and the $q$-type assumption on BN254, given a correctly generated CRS. Nullifier collision resistance follows from the collision resistance of Poseidon with 128-bit security. Domain separation via the chain identifier $\chi$ and circuit tag $\tau$ prevents cross-circuit and cross-chain nullifier collisions.

The binding between public inputs and transaction fields (e.g., `content_hash` in AnonymousPost) is enforced at the consensus layer, not in-circuit, preventing proof transplantation between transactions.

### 12.3 Encryption

Per-edge encryption with XChaCha20-Poly1305 provides IND-CCA2 security. The 192-bit nonce of XChaCha20 eliminates nonce-reuse concerns for practical volumes. Domain-separated KDF with sender and recipient identifiers prevents key reuse across conversations.

Forward secrecy is achieved through per-message ephemeral keys (Section 6.4): each edge and each DM is encrypted with a fresh ephemeral X25519 key pair, so compromising the recipient's long-lived DM private key does not reveal past messages; the ephemeral secrets no longer exist. This provides forward secrecy in the IND-CCA2 sense but not post-compromise security: after a key compromise, the attacker can decrypt *future* messages encrypted to the same static key until the key is rotated via a new `PublishDmKey` transaction. A Double Ratchet mechanism (providing both forward secrecy and post-compromise security) is incompatible with the protocol's asynchronous, store-and-forward delivery model, where messages may arrive out of order from the DA layer.

### 12.4 Recovery Security

Social recovery requires a 7-day timelock after threshold guardian approval. Timelock recovery enforces a minimum 3-day waiting period. Both mechanisms prevent instant account takeover. Reconfiguration (key rotation, guardian removal, recovery-config changes) is unconditionally blocked during any active recovery, preventing an attacker from silently disabling the recovery mechanism.

**Cancellation policy.** The owner key and a stolen key are indistinguishable on-chain, creating a fundamental tension: if the active key can cancel recovery during the timelock, an attacker who steals the key vetoes every legitimate recovery attempt; if it cannot, a malicious guardian coalition completes an irrevocable takeover while the legitimate owner watches. The protocol resolves this via a per-account `CancelPolicy` enum set in `SetSocialRecovery` (changeable only through a timelock): *KeyWins* (default) permits `CancelRecovery` during the waiting period, recommended when guardian collusion is the primary threat (FROST provides the key-theft fallback); *QuorumWins* rejects `CancelRecovery` once the timelock begins, protecting legitimate recovery from key-thief interference at the cost of exposure to guardian-initiated takeover. Guardian-set changes are themselves subject to a timelock regardless of policy, preventing an attacker from removing guardians before initiating recovery. FROST threshold recovery is unaffected: it completes atomically without a timelock or cancellation surface.

Threshold (FROST [6]) recovery imposes no timelock, as the security derives from the threshold: an attacker must compromise $t$ out of $n$ key-share holders.

### 12.5 State Integrity and DoS Resistance

All state transitions are deterministic. Borsh serialisation, `BTreeMap` ordering, checked arithmetic, and the absence of floating-point ensure that all validators arrive at the same state root given the same block sequence. Note that this guarantee applies to validators that have processed the same block history. Newly joining validators support two sync modes. *Block sync* (default) fetches blocks in batches of up to 64, validates each (parent hash continuity, QC, state root), and replays state transitions sequentially. *State sync* activates automatically when a fresh node (height = 0) discovers a peer ≥256 blocks ahead. The node requests a full state snapshot via `/asgp/state-sync/1` (binary,

length-prefixed, max 64 MiB). The responder serialises the block header at its current height plus all key-value entries across the 21 state-root-included column families (Section 11.2); the 4 node-local families (metadata, sync state, oracle cache, temporary indices) are excluded from the snapshot as they contain per-node data that would cause unnecessary traffic and potential state divergence. The receiver verifies the header's QC against the genesis validator set, loads all entries, recomputes the state root using the same compound-key algorithm as `apply_block` (`cf_len` || `cf_name` || `key`), and aborts if the computed root diverges from `header.state_root`. After snapshot load the node switches to block sync for the remaining blocks. Any state-sync failure falls back to full block replay from genesis.

DoS resistance mechanisms:

- Block size cap (256 transactions) bounds per-block computation.
- Nonce ordering prevents mempool flooding.
- View jump limit (256 views) prevents fabricated high-view QCs.
- Capacity metering rate-limits sponsored accounts.
- Deposit requirements (labeler: 10,000; schema: configurable) prevent Sybil-scale metadata abuse.
- Per-account hard caps (credentials: 64, derived keys: 32, retired keys: 16) bound state growth.

**Admission-time capacity enforcement.** Capacity metering (Algorithm 4) is enforced at three points: mempool admission (transactions from accounts without an active sponsor, or whose sponsor lacks quota or CC balance, are rejected before entering the pool — at the RPC boundary and on gossip receipt), batch construction (re-checked against current state during block proposal, as the sponsor may have exhausted capacity or deactivated since admission), and runtime execution (consensus-level enforcement). Without admission-time enforcement, unsponsored transactions could occupy block capacity at zero cost before being rejected at execution, displacing fee-paying traffic. The admission check is read-only and simulates epoch replenishment against a copy of the provider state; state-dependent admission failures on gossiped transactions do not incur peer penalties, as they are not evidence of peer misbehaviour. Sixteen transaction types are sponsorship-exempt: account creation, key rotation (a security response that must remain available when a sponsor is exhausted), recovery execution, provider lifecycle operations, anonymous content anchoring, oracle submissions, and governance. Recovery *configuration* (`SetSocialRecovery`, `SetTimelockRecovery`) is not exempt: it is performed while the sponsor is active; accounts whose sponsor has been exhausted retain full recovery *execution* and key rotation, and regain configuration ability upon migration to a new provider.

### 12.6 Data Availability

Proof of publication is provided by the quorum certificate: validators verify blob retrievability before voting, so a committed block with a valid QC ($> 2S/3$ stake) attests that the referenced data was available at commit time.

Continued availability relies on two mechanisms. Erasure coding distributes each blob as $N$ fragments across distinct providers, tolerating up to $N - K$ simultaneous provider failures. Provider diversity across jurisdictions raises the collusion threshold for coordinated data suppression. A fisherman challenge protocol (challenge-response with slashing for non-production) could strengthen these guarantees, but faces the fisherman's dilemma: on-chain proof of withholding is impossible because a challenger cannot prove that data was never served, only that it was not served to the challenger. Fisherman challenges are therefore deferred to future work and are *not* part of the current security model.

Validators perform chunk sampling with Merkle proofs: each node randomly selects 16 chunks (seeded by blob hash and parent hash), fetches them via the chunk protocol, and verifies Merkle inclusion against the erasure-coded root. With 16 samples and 50% chunk availability, detection probability exceeds $1 - 2^{-16} > 99.998\%$. This provides probabilistic data availability verification within the validator set, but does not extend to external light clients in the sense of Celestia's DAS; the availability guarantee is bounded by the validator set's trust assumption (Section 4).

### 12.7 Threat Model

We consider four adversary classes, each with distinct capabilities and corresponding protocol defences.

**Rational adversary.** A cost-minimising attacker who equivocates only when expected profit exceeds the 10% slash plus forgone rewards during the 7-epoch jail. The security budget (Equation 19) makes equivocation unprofitable when detection probability is high, which it is for on-chain double-signing where any honest validator can submit evidence.

**Byzantine validators ($f < n/3$ by stake).** Up to $f$ validators may deviate arbitrarily from the protocol. The safety and liveness guarantees under this assumption are established in Section 4 (Theorem 1 and the quorum intersection argument).

**Censorship and liveness attacks.** A Byzantine leader may selectively exclude transactions from proposed blocks (censorship) or refuse to propose blocks at all (liveness attack). Neither attack is currently punishable via slashing, as the protocol cannot distinguish a censoring leader from one that simply did not receive the excluded transactions. Liveness attacks are bounded by the timeout mechanism: after the view timeout expires, validators issue timeout votes and the protocol advances to the next leader within $f + 1$ views in the worst case. Censorship by a single leader is therefore temporary (one slot), but a coalition of $f$ consecutive leaders could censor for up to $f$ slots. Long-term censorship resistance relies on the assumption that honest leaders constitute $> 2/3$ of the leader schedule. Protocol-level censorship resistance mechanisms (e.g., threshold encrypted mempools, inclusion lists) are left as future work.

**Sybil attacker.** An adversary creates $M$ accounts with mutual vouching to inflate trust scores. Defences:

1. **Vouching threshold**: each account passes the Sybil check (Algorithm 3) only if vouches from high-trust accounts exceed $\theta$.
2. **Graph analysis**: indexers detect Sybil clusters via spectral methods; colluding accounts form a weakly connected subgraph with low bridging coefficients ($\beta \approx 0$).
3. **Capacity cost**: creating $M$ accounts costs $M$ capacity units (at default cost 1 each), requiring provider stake $\geq M/C_{\text{rate}}$ tokens.

Under honest-majority vouching, the Sybil Bound (Section 10.3) bounds the aggregate trust of a Sybil cluster.

**DA adversary.** Colluding storage providers may withhold data. If all DA providers for a blob disappear, the historical content becomes unavailable, but chain state (accounts, credentials, graph roots) is preserved in consensus. Economic enforcement via fisherman challenges (challenge-response with slashing) is a potential future hardening measure but is not part of the current security model due to the fisherman's dilemma: on-chain proof of withholding is fundamentally impossible. Graceful degradation: new content can still be posted and anchored; only retrieval of old content is affected until providers recover or are replaced.

### 12.8 Catastrophic Recovery

Three complementary recovery paths provide defence in depth against key compromise (Section 3.6). In practice, recovery paths may share failure modes: guardians are typically drawn from the account holder's social circle (correlated with social-engineering risk), FROST key-share holders may overlap with guardians, and backup key storage discipline correlates with primary key discipline. Users should diversify recovery paths across distinct trust domains.

Table 10: Recovery mechanism comparison.

| | **Social** | **Timelock** | **FROST** |
|---|---|---|---|
| Trigger | $t$-of-$n$ guardians | Backup key | $t$-of-$n$ shares |
| Delay | 7 days | 3 days | Immediate |
| Cancel | Per policy[a] | Per policy[a] | N/A |
| Reconfig | Blocked | Blocked | N/A |
| Risk | Guardian collusion | Key theft | Share leakage |

[a] *KeyWins* (default): owner key may cancel during timelock. *QuorumWins*: cancel blocked after timelock begins.

No automated quorum recovery exists for the validator set itself: if more than $1/3$ of stake is lost (e.g., mass key compromise), recovery requires manual governance intervention (social consensus among remaining validators to restart the chain with a pruned validator set).

## 13. Related Work

The whitepaper [22], Section 9, provides market positioning and competitive analysis. This section adds technical comparisons relevant to the specification. Table 11 compares protocol capabilities across eleven dimensions.

[1] Social edges encrypted (per-edge AEAD); vouching topology intentionally public for trust scoring. See Section 6.3.

Table 11: Comparison of decentralised social identity protocols.

| | **ULI** | **AT Proto** [19] | **DeSo** [7] | **DSNP** [8] | **Farcaster** [16] | **Nostr** [17] |
|---|---|---|---|---|---|---|
| Identity | Numeric ID, DID | did:plc | Public key | DSNP Id (u64) | ETH FID | Public key |
| Key rot. | Yes (16 ret.) | Yes (PLC op) | No | Yes | Yes (L1) | No |
| Recovery | Social, TL, FROST | PLC rotation | None | Provider | ETH-based | None |
| Content | Off-chain (DA) | Off-chain (PDS) | All on-chain | Off-chain | Off-chain | Relays |
| ZK cred. | 6 circuits | None | None | None | None | None |
| Encrypt. | Per-edge AEAD | None | None | None | None | NIP-44 |
| Agent attest. | Derived keys | App passwords | None | None | None | None |
| Token req. | No (capacity) | No | Yes | Yes | Yes (ETH) | No |
| Moderation | Labelers | Labelers | None | Provider | Channel | Relay |
| Graph priv. | Layered[1] | Public | Public | Public | Public | Public |

**Proof-of-personhood and attestation systems.** WorldCoin, Gitcoin Passport, and their limitations are analysed in the whitepaper [22], Section 9. The key technical distinction: Reclaim Protocol uses a proxy-based zkTLS architecture whose security rests on a non-collusion assumption between the HTTPS proxy (attestor) and the prover; a compromised attestor can forge arbitrary attestations and observes traffic metadata. All three systems anchor verification to external infrastructure, inheriting its trust assumptions. ULI treats external attestation as an optional credential layer (Section 3), providing graph-based Sybil resistance as a verifiable trust-scoring market (staked indexers with fraud-provable score commitments) and making strong external credentials portable and privacy-preserving via ZK proofs and a global nullifier set.

**Decentralised social protocols.** Positioning relative to AT Protocol, DeSo, DSNP, Farcaster, Nostr, and BrightID is discussed in the whitepaper [22], Section 9. The technical distinctions relevant to this specification: AT Protocol's `did:plc` relies on a single PLC directory server (no on-chain recovery); its relay architecture does not provide cryptographic anchoring of content availability. DeSo shares Fast-HotStuff consensus but stores all content on-chain. Farcaster requires Ethereum L1 gas for key rotation; social data resides in off-chain hubs without cryptographic anchoring. BrightID's trust algorithms (SeedConnected, Bitu, Aura) operate on a publicly visible graph, making privacy-preserving scoring impossible. The protocols described above do not provide ZK credentials, per-edge encryption, or protocol-level key recovery.

## 14. Test Coverage

The implementation is covered by an automated test suite spanning all 16 crates, including property-based fuzzing of transaction admission and consensus safety invariants, and end-to-end tests of the recovery and provider lifecycles. The suite runs in CI on every commit.

## 15. Conclusion

This specification defines the ULI protocol across five layers: (1) key-independent identity with three recovery mechanisms (social, timelock, FROST threshold); (2) Fast-HotStuff BFT consensus with BLS12-381 aggregation and a 2-chain commit rule achieving $O(n)$ message complexity; (3) six Groth16/BN254 ZK circuits with Poseidon commitments and domain-scoped nullifiers; (4) an encrypted social graph with per-edge AEAD and dual Merkle roots (Blake3 for integrity, Poseidon for in-circuit proofs); and (5) a capacity-staking model with fiat-pegged Capacity Credits and permanent fee burn proportional to network usage.

The specification is implementation-complete: all 49 transaction types, six ZK circuits, the DA layer with three storage backends, the stake-weighted median oracle with TWAP smoothing, on-chain governance, provider exit lifecycle, ZK rate limiting, incremental sparse Merkle tree, and state sync are realised in the codebase. Open engineering work includes recursive proof aggregation for ZK throughput scaling, chunked state-sync streaming for mainnet state sizes, and the Groth16-to-PLONK migration enabled by circuit versioning (`RegisterCircuit`/`DeprecateCircuit`). For market context, economic rationale, and competitive positioning, see the companion whitepaper [22].

## References


[1] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. ACM PODC*, pages 347–356, 2019.

[2] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai. Fast-HotStuff: A fast and resilient HotStuff protocol. *IEEE Trans. Dependable and Secure Computing*, 21(4):2478–2493, 2023.

[3] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology, ASIACRYPT 2001*, pages 514–532. Springer, 2001.

[4] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology, EUROCRYPT 2016*, pages 305–326. Springer, 2016.

[5] L. Grassi, D. Kales, R. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, 2021.

[6] C. Komlo and I. Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In *Selected Areas in Cryptography (SAC)*, pages 34–65. Springer, 2020.

[7] DeSo Foundation. DeSo: The decentralized social blockchain. White paper, 2022.

[8] Unfinished Labs. DSNP: Decentralized social networking protocol specification. Version 1.2, 2023.

[9] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography, PKC 2006*, pages 207–228. Springer, 2006.

[10] M. Hamburg. Decaf: Eliminating cofactors through point compression. In *Advances in Cryptology, CRYPTO 2015*, pages 705–723. Springer, 2015.

[11] S. Shahaf, E. Duffy, S. Kalodner, and M. Naor. Who watches the watchmen? A review of subjective approaches for Sybil-resistance in proof of personhood protocols. *Frontiers in Blockchain*, 2020.

[12] V. Buterin. What do I think about biometric proof of personhood? Technical note, 2023.

[13] E. Pariser. *The Filter Bubble: What the Internet Is Hiding from You*. Penguin Press, 2011.

[14] C. A. Bail, L. P. Argyle, T. W. Brown, et al. Exposure to opposing views on social media can increase political polarization. *Proc. Nat. Acad. Sci.*, 115(37):9216–9221, 2018.

[15] F. P. Santos, Y. Lelkes, and S. A. Levin. Link recommendation algorithms and dynamics of polarization in online social networks. *Proc. Nat. Acad. Sci.*, 118(50), 2021.

[16] Farcaster Protocol. Farcaster protocol specification. 2023.

[17] Nostr Contributors. Nostr: Notes and other stuff transmitted by relays. NIP-01, 2023.

[18] P. Wuille. BIP-340: Schnorr signatures for secp256k1. Bitcoin Improvement Proposal, 2020.

[19] Bluesky PBC. AT Protocol: The social internet. Protocol specification, 2024.

[20] W3C. Decentralized identifiers (DIDs) v1.0. W3C Recommendation, 2022.

[21] W3C. Verifiable credentials data model v2.0. W3C Recommendation, 2024.

[22] ULI. ULI: Sovereign identity with zero-knowledge credentials. Whitepaper, 2026.

[23] Al-Bassam, M., Sonnino, A., and Buterin, V. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 2018.