The Stamp Protocol

Secure Trustless Anonymous Micropayment Protocol (S.T.A.M.P.)

Technical Whitepaper version: 0.1 (Draft)
Date: September 2025

Author: Joe, Stamp Protocol
Contact: joe@stampprotocol.org

Abstract

On Solana today, executing actions—whether on-chain (sending a message, placing a trade, joining a DAO vote) or off-chain (API queries, RPC calls)—always ties back to a wallet. Direct transfers expose identity and usage patterns, while even “anonymous” accounts leak metadata: for example, a supposedly private messaging account may reveal its owner’s timezone simply by the hours when messages are sent. Off-chain accounting avoids the chain but reintroduces trust and manipulation risks.

The Stamp Protocol solves this by breaking the link between funding and usage. Instead of paying with a wallet each time, users prepay once and receive digital “stamps” that act as one-time credits. Before using them, these credits can be privately exchanged for fresh ones — like handing old stamps to a blind cashier who only checks they’re real, then passes back brand-new stamps. Once swapped, the new stamps carry no visible link to the originals, so when they are finally spent, nobody can trace them back to the buyer.

The Stamp Protocol (S.T.A.M.P.) implements this model on Solana as unlinkable prepaid credits called *stamps*. Each stamp is a one-time bearer token defined by a short cryptographic secret: small enough to pass around like a token, strong enough to prevent forgery, and valid only once when redeemed. Stamps can be spent directly or transferred to others, circulating like digital bearer notes.

To guarantee soundness, zero-knowledge proofs ensure balance integrity when stamps are first issued. Redemptions are then handled through relayers under an optimistic model: services execute immediately, while watchers have up to 30 days to challenge fraud. This dual design keeps redemptions fast and final for users, while ensuring durable accountability for the system.

S.T.A.M.P. turns Solana into the first high-throughput chain where value moves with the speed of finance, but with the privacy of stamps—making every service, from trading to messaging, instantly monetizable without ever exposing who’s paying.

Table of Contents

1. Introduction

The Stamp Protocol is motivated by the fact that Solana has no built-in way to support anonymous prepaid usage. While the network supports fast, cheap transfers, direct payments are poorly suited to services that require unlinkability between payer and usage. Each payment exposes the payer's identity, leaves a permanent trail that links account to activity, and makes private access to services impossible. Existing off-chain workarounds shift trust to the operator and break composability.

S.T.A.M.P. solves this by reintroducing the metaphor of postage: credits are bought once, in bulk, then consumed one by one without linkability. Each credit, or stamp, is defined by a **16-byte secret**, small enough to pass around like a token yet strong enough to resist brute force. Possession of this secret is sufficient for redemption: the protocol derives a nullifier, checks it has not been seen before, and enforces one-time spendability. A stamp thus functions like a cashable bearer note or one-time traveler's cheque — whoever holds the secret can redeem it, exactly once. Crucially, stamps can also be **reshaken**, transferring their value into a fresh 16-byte secret and invalidating the old one. This lets ownership move freely while preserving unlinkability, with no wallet state or persistent account required.

1.1 Prior Art & Novelty

On-chain payments on Solana today are dominated by two extremes: direct token transfers, which expose payer identity and usage patterns, and off-chain accounting schemes, which rely on trust and are vulnerable to manipulation. Existing privacy systems (e.g., Tornado Cash, Zcash) focus on shielding account-based value transfers, but they still assume balances tied to identities. They do not provide prepaid, unlinkable usage credits that can be spent directly without maintaining an account.

The Stamp Protocol introduces a different model, combining five properties not found together elsewhere:

1. **Accountless prepaid credits** — users lock funds once, then spend unlinkable 16-byte secrets later; no wallet state or persistent account is needed.
2. **One-time spendability** — each secret is consumed exactly once, with a nullifier enforced on-chain to prevent reuse.
3. **Unlinkable transferability** — value can be "reshaken" into new secrets, breaking any link to the original payer while preserving spendability.
4. **Built-in accountability** — every issuance and redemption is anchored into rolling Merkle pyramids, enabling fraud detection across a 30-day horizon without growing on-chain state.
5. **Risk-backed security** — relayers and aggregators front execution costs and bond capital; misbehavior shifts losses onto them, while watchers are rewarded for proving fraud.

The Secure Trustless Anonymous Micropayment Protocol (S.T.A.M.P.) is, to our knowledge, the first system that unifies these properties into a practical credit layer for Solana. It enables per-use charging without revealing payer identity, achieves high throughput with lightweight proofs, and retains decentralization through open relayer and watcher markets.

2. System Overview

The Stamp Protocol is maintained by four types of actors, each with a distinct responsibility:

- **Users** purchase credits and generate stamps locally. They submit commitments and payments on-chain.
- **Aggregators** observe all issuances, build per-epoch Merkle trees, and submit roots on-chain.
- **Relayers** accept spend proofs and secrets from clients, perform validity checks, and then execute the requested service calls (on- or off-chain) before redeeming the credit value.
- **Watchers** monitor aggregator and relayer behavior. They can challenge invalid roots or spends, earning slashed collateral as reward.

2.1 Users

At issuance time, a user prepays credits by sending SOL to the Stamp Protocol program on Solana. In the same transaction, the user submits a batch of cryptographic commitments — hash-like receipts that hide the details of each credit — together with a zero-knowledge proof.

The proof shows that the commitments are well-formed and that their combined value equals the transferred payment amount. The on-chain program verifies this proof and checks that the lamports transferred are at least the declared sum_total. It then extends a global rolling hash H, which acts as an immutable receipt of issuance, binding the sequence of commitments without storing them individually.

Each commitment corresponds to what we call a **stamp**: a prepaid credit unit defined by a short secret and its value binding. A stamp can later be demonstrated in two ways:

- by revealing the secret directly, which suffices for spending, or
- by producing a zero-knowledge proof that attests to knowledge of the secret and its correct binding, without exposing the secret itself.

Either path convinces a verifier that the stamp was legitimately issued and unspent, while preserving unlinkability between the original payer and eventual redemption.

2.2 Aggregators

Aggregators continuously observe issuance submissions. Valid commitments are inserted into a Poseidon-based Merkle tree. At the end of each epoch, the aggregator produces that epoch's root and folds it upward into the five-level rolling Merkle structure maintained by the protocol.

This hierarchical tree accumulates recent epoch roots into progressively higher layers. The resulting combined root, representing all outstanding stamps, is published on-chain.

To incentivize liveness, the first aggregator to submit a valid root for a pending range is paid a fixed protocol fee per entry sealed. This guarantees that roots continue to advance even if some aggregators go offline, and creates an open, competitive market for aggregation.

2.3 Relayers

Redemption begins when a client presents a stamp secret to a relayer. The relayer reconstructs the corresponding commitment, derives its nullifier, and checks locally that it has not already appeared in the spent set.

If valid, the relayer executes the requested service call — either an **on-chain transaction** (covering the Solana fee from its own wallet) or an **off-chain action** such as forwarding a message or serving an API response. The relayer is then reimbursed with the full credit value:

- If an on-chain fee was paid, the margin is retained as profit.
- If the service was off-chain, the entire stamp value becomes the relayer's payment.

This flexible model balances costs and incentives without requiring fixed fee tiers at issuance.

2.4 Watchers

Watchers continuously monitor relayer behavior. They audit redemption trails and nullifier sets. If they detect invalid commitments or double spends they can challenge by submitting compact Merkle proofs on-chain.

When a challenge succeeds, the dishonest relayer's unvested payouts are slashed to restore the pool, and their bonded stake is transferred as bounty to the watcher. This guarantees strong economic incentives for honest participation while preserving unlinkability between issuance and redemption.

2.5 Design Asymmetry

The protocol deliberately front-loads computation at issuance, where commitments are sealed and zero-knowledge proofs validated. Redemption, by contrast, requires only lightweight secret checks, so spends finalize quickly and cheaply. This **asymmetry** makes buying credits more expensive, but ensures that frequent redemptions scale efficiently at Solana throughput.

3. Commitment Anchoring Pyramid

To hold issuance, nullifiers, and relayer events accountable without bloating on-chain state, the Stamp Protocol compresses all activity into a hierarchical Merkle structure. This anchoring system allows high-volume data to be retained off-chain while still being provable against a small rolling set of roots on-chain. Every event can thus be verified against a durable anchor, keeping costs predictable while ensuring long-horizon accountability.

The design is organized as a layered pyramid with fixed arity. The base unit is the Solana slot (≈400 ms). At the lowest level (L0), raw commitments from each slot are aggregated off-chain into slot-level Merkle trees. These slot roots are rolled upward into progressively higher layers, which accumulate history across longer horizons. The top layer, L4, anchors approximately thirty days of history in a single root committed on-chain once per minute. We choose a branching factor of 8 because it balances proof size and update cost.

3.1 Parameters

Each level is defined by three parameters: **granularity, arity, and depth**. The protocol fixes arity at eight, balancing proof size against update overhead. Updates occur once per minute, which corresponds to ~150 slots, and the retention horizon extends to ~30 days.

3.2 Levels

- **L0 (local mirrors):** Each slot (~400 ms) collects raw commitments and batches them into a Merkle tree of arity 8. The root exists only off-chain in aggregator mirrors. Depth = $\log_8(n)$, where *n* is commitments per slot.

- **L1:** Aggregates 150 L0 roots → ≈1 minute window. Depth = 3.

- **L2:** Aggregates 60 L1 roots → ≈1 hour window. Depth = 2.

- **L3:** Aggregates 24 L2 roots → ≈1 day window. Depth = 2.

- **L4 (top):** Aggregates 30 L3 roots → ≈30 days window. Depth = 2.

Each layer is implemented as a ring buffer of fixed width. As new entries arrive, the oldest is evicted and the parent root recomputed in place.

3.3 Rolling Hash Anchors

Before commitments are sealed into the pyramid, they are first chained into a global rolling hash. Each issuance extends the sequence: H_next = Poseidon(H_prev || commitments_batch).

This rolling hash provides an immutable, append-only log of all commitments in strict order. Rather than storing every individual commitment on-chain, the protocol records only the evolving hash, which acts as a compact receipt.

When an aggregator seals a batch, it defines a contiguous range of the rolling hash, from **H_start** to **H_end**. The corresponding commitments in that range are placed into an L0 → L1 tree, and a zero-knowledge proof is provided that:

- the commitments indeed extend H_start to H_end with no gaps or reordering, and
- the resulting Merkle root faithfully represents that exact batch.

Hash ranges can be chained flexibly: a submission may cover **$H\_start_1 \rightarrow H\_end_1$**, or bundle multiple contiguous ranges (e.g., **$H\_start_1 \rightarrow H\_end_4$**) into a single proof. What matters is that the start and end anchors match the published rolling hash sequence.

The rolling hash therefore serves two purposes:

1. **Ordering**: It fixes the canonical sequence of commitments, so no aggregator can omit or reshuffle entries.
2. **Bridging**: It provides the link between raw commitments and the Merkle pyramid, ensuring that every sealed L1 root corresponds exactly to a prefix of the rolling hash.

This design allows the pyramid to remain compact while still covering the full history: every L1 root corresponds to a proven range [H_start, H_end], and higher layers (L2–L4) inherit that integrity without re-checking commitments individually.

3.4 Update Flow

Every minute, the protocol performs a bounded update sequence. A new L1 root is created from the last 150 slots and inserted into the L1 buffer. This cascades upward: once L1 rolls over, L2 is updated; when L2 rolls, L3 is updated; and when L3 rolls, L4 is updated. The current L4 root is committed on-chain as the canonical anchor.

Indexing is modular: the position within each buffer is given by simple modular arithmetic (e.g., $i_2 = t \bmod 160$ for L2). This guarantees O(1) sliding updates regardless of throughput (L0-L4).

3.4.1 Sparse periods & backfilling

If no submissions occur for one or more minutes, aggregators backfill those minutes off-chain by inserting **EMPTY** L1 leaves for each missed minute and rolling them up through L2–L4. When activity resumes, a single on-chain update publishes the next L4 root with a strictly increasing epoch index. This keeps the time grid consistent without requiring one on-chain transaction per idle minute; watchers can recompute the same backfilled roots from public data.

3.5 Efficiency

The pyramid is designed to minimize both on-chain and off-chain overhead. Aggregators bear the heavy cost only once, when sealing an L1 root against the rolling issuance hash with a zero-knowledge proof. From that point onward, the Stamp contract maintains the higher layers (L1–L4) internally. Each new L1 root triggers at most nine Poseidon rehashes inside the program, a fixed cost of ≈9k compute units per minute — negligible at Solana scale. Proofs remain compact: in the worst case, a zero-knowledge circuit needs to carry about 63 sibling hashes to connect an L1 leaf to the canonical L4 root. This bounded overhead ensures that inclusion proofs remain small, on-chain work remains predictable, and throughput scales independently of overall issuance volume.

3.6 Properties

This structure achieves several key properties:

- **Completeness:** every slot is anchored into the rolling root, ensuring nothing can be excluded.
- **Durability:** the L4 buffer retains 30 days of history, enabling long-horizon challenge windows.
- **Privacy:** proofs are always referenced against the L4 root, so the precise leaf index is hidden inside the ZK circuit.
- **Efficiency:** updates are strictly bounded, requiring a fixed nine hashes per minute, while on-chain state remains constant.

The Merkle root pyramid thus serves as the backbone of the protocol's accountability model: it ties together high-frequency events at the slot level with durable long-horizon commitments, while keeping both computation and storage costs under control.

4. Stamp Issuance

When a user prepays for credits, they generate a batch of stamps locally. A stamp is a small, transferable bearer instrument consisting of a random secret and an associated value. Only commitments are published at issuance; the underlying secrets remain off-chain until redemption. When a stamp is spent, the holder reveals its secret. Prior reshaking ensures this disclosure cannot be linked to the funding account

4.1 Issuance Overview

Stamp issuance is a two-step process:

1. **Commitment stage.** The user posts a batch of commitments together with a zero-knowledge proof that their declared sum matches the prepaid funds. These commitments extend the rolling hash on-chain and are appended into the current epoch's global tree. At this stage, they are recorded but not yet redeemable.

2. **Activation stage.** Through a relayer, the user reshakes these commitments into fresh leaves in the global tree. The original epoch entries are nullified, and the reshaken leaves become the live, redeemable ones.

This separation of stages ensures unlinkability between funding accounts and live credits. On chain proof validations are required only twice in the lifecycle of a stamp — once when commitments are first issued, and once again when they are reshaken into fresh leaves. After activation, redemptions require only the short secrets, making spends lightweight while still unlinkable to the payer.

4.2 Commitment stage

Commitment and sealing is the first stage of issuance. In this step, SOL is transformed into cryptographic commitments and anchored to the chain via the rolling hash. Aggregators seal these commitments into Merkle roots, ensuring order and integrity without exposing the underlying secrets.

4.2.1 User side

The user's device generates a fresh 16-byte random secret *t* for each stamp. Each stamp is assigned a value class (e.g. 0.001 SOL, 0.1 SOL, etc.) from the denomination table. In a zero-knowledge circuit, the device derives a 16-byte commitment hash for each stamp and produces a Groth16 (BN254 curve) proof that their values sum exactly to the prepaid amount. The resulting commitments, together with the proof and its public inputs, are then submitted to the on-chain contract.

4.2.2 On-chain contract

Upon receiving the commitments, their accompanying proof, and the public inputs, the Stamp contract verifies the zero-knowledge proof to ensure that each commitment is well-formed and that the declared *sum_total* is correct. The contract then checks that the lamports transferred equal *sum_total*, and transfers those lamports into the central credit pool (the program vault) from which redemptions will later be paid. This ensures that every stamp is fully prepaid and that future redemptions are always covered by escrowed funds.

If verification succeeds, the commitments are appended to the global rolling hash:

H_next = Poseidon(H_prev || commitments_batch).

This rolling hash serves as an immutable issuance receipt, binding the sequence of commitments without storing them individually. It defines ordered "hash ranges" between checkpoints. Each range must eventually be sealed by submitting a Merkle root with a proof that exactly covers the commitments from H_start to H_end. Sealing is **permissionless**: any party may submit a valid root proof for a pending range, and the first valid submission is accepted. Aggregators who perform this sealing are compensated with a protocol fee (§4.4). This ensures liveness even if some aggregators go offline. Multiple consecutive ranges can be bundled into one submission (e.g., $H\_start_1 \rightarrow H\_end_4$), but they must follow the rolling order without gaps.

To keep order consistent while allowing parallel aggregation, the contract enforces a bounded lag policy. Root submissions may arrive out of order, but if a range *N* remains unsealed while a later range *N+Δ* has already been sealed, no further submissions are accepted until *N* is provided. This prevents permanent gaps while still letting multiple aggregators work in parallel within a bounded window.

When multiple submissions arrive in the same slot, the contract applies a deterministic tie-breaking rule:

1. First sort by slot_number (ascending).
2. For submissions with the same slot, sort by the lexicographic order of H_start.

This ensures a canonical ordering without ambiguity. Late submissions can always be placed correctly once they arrive, and everyone derives the same global sequence.

To store and verify these sealed roots efficiently, the protocol uses the slot-based Merkle pyramid described in §3. Each sealed root becomes an L1 leaf, which rolls upward into higher layers (L1–L4). Only L4 is persisted on-chain, giving a single canonical root that represents ~30 days of history.

4.2.3 Aggregator side

Off-chain, aggregators observe issuance submissions and collect new commitments. At chosen intervals, they seal a contiguous hash range (H_start → H_end) into an L1 tree. Inside a zero-knowledge circuit, they prove two things: (i) the commitments extend H_prev to H_end with no gaps or reordering, and (ii) those same commitments are placed in strict order as leaves of the L1 Merkle tree. The circuit outputs both the updated rolling hash H_end and the L1 root.

The aggregator submits this result to the Stamp contract. The contract verifies the proof; if valid, it marks the covered range as sealed, removes it from the pending set, and records the resulting L1 root. That root is then incorporated into the Merkle pyramid (§3), ensuring continuity of the global structure. As compensation, the aggregator receives a fixed protocol fee per sealed entry, providing direct economic incentive to keep ranges up to date.

Once sealed, the corresponding **L1 roots (≈150 per minute)** are retained on-chain. The full L1 trees that produced them are kept off-chain by aggregators (and any independent mirrors) for the duration of the 30-day validity horizon, so that inclusion paths can be served on request. Because both the commitments and the rolling hash sequence are public, anyone can reconstruct these trees independently and provide equivalent proofs, ensuring redundancy and decentralization.

Users (or relayers) later prove inclusion against the global root by referencing R_top_curr (L4 root) or R_top_prev (previous L4 root) and supplying the fixed 32-sibling Merkle path from L0→L4; the L1 root itself is already ZK-linked to the rolling hash at issuance, preserving end-to-end consistency without re-verifying large trees on-chain.

4.3 Activation stage

Activation is the second stage of issuance, where sealed commitments are converted into live, spendable credits. Through a relayer, the user reshakes previously sealed leaves of the global tree into fresh commitments. The original leaves are nullified, and a zero-knowledge proof is submitted demonstrating knowledge of their secrets, correctness of their values, and that the total value of the new commitments is equal to, or less then, the total value of the consumed ones. The difference is realized as the relayer's execution fee. Once accepted, the reshaken leaves are treated as newly issued, inheriting the full 30-day validity horizon and becoming indistinguishable from any other active credits in the system.

4.3.1 User side

To activate, the user proves knowledge of the secrets and values (e.g. amt_byte) tied to previously sealed commitments in the global tree. For each consumed leaf, the user obtains its Merkle path (e.g. via a relayer API operating behind Tor, or from a local mirror) and inputs it into a zero-knowledge circuit.

Inside the circuit:

- The paths verify that the leaves are correctly included under the published global root.
- The declared values are summed and checked for consistency.
- New commitments are derived from this total, ensuring they do not exceed the consumed value.

The output is a Groth16 proof, the new commitments, and the nullifiers of the consumed leaves. These artifacts are then handed off to a relayer.

4.3.2 Relayer side

The relayer receives the user's Groth16 proof, commitments, and nullifiers, verifies their validity, and submits them to the Stamp contract in a transaction. To do so, the relayer prepays both the Solana transaction costs and the full credit value being reshaken. The Stamp contract immediately instantiates the fresh commitments for the user, while the matching reimbursement is drawn from the central credit pool and deposited into the relayer's Payout PDA under a vesting delay (e.g., 24 hours). The reimbursed amount equals the total value of the consumed stamps; the new commitments may sum to a slightly smaller value. The difference — the residual — covers the relayer's transaction costs and provides their profit margin.

During this window, watchers can challenge invalid activations. If a challenge succeeds, the relayer's collateral is slashed, the pool is restored, and any surplus is awarded to the challenger. This ensures user actions are irreversible and the credit pool remains solvent; only the relayer ever bears financial risk. Because activations are submitted via Tor or equivalent anonymization, they are unlinkable to the user and appear to the relayer as generic nullify-and-reshake operations.

Note. The Merkle paths, secrets, and their associated amt_bytes remain private inside the proof circuit and are never re-published on-chain. Only the new commitments and the nullifiers of the consumed leaves appear externally. This ensures double-spend protection while preserving unlinkability between the original payer and the active credits.

4.3.3 On-chain contract

The relayer's transaction includes the Groth16 proof (~200 bytes), the nullifiers of the consumed leaves, and the new commitments. The protocol fixes Groth16 over the BN254 curve, since this pairing-friendly curve admits efficient on-chain verification and compact proofs. Solana programs can verify a single BN254 Groth16 proof within ~200k compute units, making it feasible as long as activations are batched.

Given Solana's transaction byte budget, ~550–600 bytes remain after the Groth16 proof and nullifiers, allowing ~30–40 new 16-byte commitments per activation. The proof enforces that each published nullifier is correctly derived from a consumed secret; the contract then accepts the activation optimistically, recording the event and updating the nullifier/commitment pyramids (§3). During the vesting window, watchers can challenge with inclusion/non-inclusion proofs; if a challenge succeeds, the relayer's stake/vault is slashed and the pool is restored. From this point forward, redemptions require only the short secrets, with no residual linkage to the payer.

4.3.4 Watcher side

Watchers continuously monitor activations to ensure that no invalid commitments or nullifiers slip through. Using the rolling Merkle pyramids (§3), they can independently reconstruct both the commitment and nullifier sets for the entire 30-day horizon. Upon each activation, watchers:

1. **Validate the Groth16 proof** off-chain using the published commitments and nullifiers. Any malformed proof or incorrect derivation is immediately detectable.

2. **Prove double spend**: to demonstrate fraud, a watcher submits an on-chain proof that a "new" nullifier already appears in the nullifier Merkle pyramid. This establishes a double-spend attempt with cryptographic finality.

3. **Audit inclusion paths**: for any suspicious activation, watchers recompute the Merkle path from the consumed leaf to the published root. A mismatch shows that the relayer forwarded an invalid proof.

4. **Challenge on-chain**: if fraud is detected, a watcher submits a challenge transaction, referencing the relevant L5 root and providing the minimal inclusion/non-inclusion proof.

Challenges are verified deterministically on-chain. If upheld, the fraudulent value is returned to the pool and the relayer's PDA is slashed; the watcher receives the residual bounty. Because activations are accepted optimistically, this challenge mechanism ensures that relayers operate under continuous risk: their profits remain at stake for the entire vesting horizon, and any attempt to push invalid activations results in capital loss.

Hygiene note. For efficiency, issue a small number of higher-value commitments at the commitment stage, then split only at activation. To harden unlinkability, add decoys: reshape the spend into many uniform-value outputs (e.g., 30 × 0.0001 SOL) plus a residual "change" commitment. Since only reshaken leaves appear on-chain (secrets and amt_bytes stay inside the proof), observers can't reliably link a funding payment to a particular activation or infer original denominations.

Re-activation. The same process also serves as a re-activation mechanism when credits near expiry. Users can reshake expiring leaves into fresh ones, thereby extending validity for another full 30-day horizon without revealing their secrets.

4.4 Stamp Commitment Definitions

A stamp encodes not only its prepaid credit value but also optional service binding metadata. This ensures that when the stamp is redeemed, the Solana program call can be enforced atomically.

4.4.1 Fields

- **Secret (t)**: A uniformly random 16-byte string generated locally. This is the core of the stamp's unpredictability.

- **Tag (TAG_CT)**: A fixed domain-separator constant preventing collisions between different uses of Poseidon in the system.

- **Amount byte (amt_byte)**: A one-byte encoding of the stamp's credit value.

  - Low 5 bits = *amt_class*, one of 32 predefined denominations (appendix B).

  - High 3 bits reserved for extensions.

- **Service binding (service_tag, optional)**: If the stamp is locked to a specific service call, a compressed 16-byte tag encodes (TAG_SVC, *program_id*, *method_sel*, *args_hash*).

4.4.2 Commitments

C_t = Poseidon(TAG_CT ‖ t) C_v = Poseidon(TAG_CV ‖ amt_byte | table_commitment | service_tag?)

L = Poseidon(TAG_LEAF | C_t | C_v)

N = Poseidon(TAG_NULL | t)

- **L** is the leaf commitment that becomes active in the dense Merkle tree of commitments.
- ***N*** is the nullifier ensuring one-time spendability; once revealed, it is inserted into the spent set.
- **table_commitment** is the 16-byte truncated hash of the denomination table. It identifies which denomination schedule the stamp belongs to, while remaining compact.

4.4.3 Enforcement

- Issuance: the program accepts whatever table_commitment the client used. If it refers to a deprecated table, the stamps will later be unredeemable, effectively burning funds.
- Redemption/reshake: the contract verifies that the table_commitment is still in the registry and not marked expired (Appendix B).

4.4.4 Relayer fee

The protocol does not encode a fixed fee inside the stamp. Instead, the residual after service execution becomes the relayer's fee, allowing fees to adapt dynamically to network conditions.

4.5 Transferability

Stamps are transferable bearer instruments. To transfer a stamp, the sender shares the stamp secret *t* (16 bytes). The recipient can reconstruct all commitments and the nullifier from *t* and generate a valid redemption.

- *amt_byte*: Optionally shared. Since *amt_class* occupies only 4 bits (16 values), the recipient can brute-force trial classes and select the one that verifies.
- service_tag: If present, it can be recomputed deterministically by client software, since it depends only on public service metadata.

No on-chain reassignment is required. Possession of *t* is sufficient to redeem. The first party to spend successfully consumes the stamp; later attempts with the same nullifier are rejected.

4.5.1 Bootstrap bundles

For practical sharing, a sender may combine immediacy and longevity — e.g. transfer one medium stamp (0.01 SOL) plus several small ones (10 × 0.001 SOL). The small credits can be spent immediately, while the larger one remains available for later use or reshaking.

4.5.2 Reshake-based transfer

Beyond raw secret sharing, the reshake mechanism also enables structured transfers. A sender can prove knowledge of existing commitments and provide new commitments provided by the recipient. The zero-knowledge proof ensures that the total value is preserved (minus any agreed fee), and the reshaken leaves are inserted directly into the global tree under the recipient's control. This creates a true transfer of credits, rather than a “first-spend-wins” race, and allows recipients to receive unlinkable fresh commitments instead of reusing the sender’s originals.

Hygiene note. For unlinkability, stamps should only be transferred by secret once they have been activated (reshaken into the global tree). Sharing raw issuance secrets before activation risks correlating the stamp to its original funding transaction. By contrast, reshake-based transfers are safe at any stage, since they generate fresh commitments for the recipient without exposing the original secrets or funding link.

5. Stamp redemption

Redemption is the process of consuming a prepaid stamp to pay for a service. A valid redemption requires that the stamp was legitimately issued, has not been spent before, and that its value is properly transferred to the relayer executing the requested service. Validation follows an optimistic model: the chain records the redemption immediately, the relayer pays the execution costs up front, and watchers are given a challenge window to contest invalid spends. If no valid challenge is raised, the redemption finalizes and the relayer's payout is settled.

Relayers are the sole party at risk in redemption. Every spend is prepaid by the relayer: the user's service call executes atomically and is irreversible, whether on-chain (via CPI) or off-chain. The corresponding credit value and relayer fee do not vest immediately; instead they accumulate in the relayer's stake vault PDA, where balances remain subject to a vesting delay (e.g. 24 hours). Within this period, any fraudulent redemption proven by a watcher results in automatic reassignment of the unvested balance to the challenger.

Once profits vest, they may be withdrawn, but relayers who retain funds in the stake vault remain exposed: watcher challenges may be raised at any point during the full 30-day validity horizon of a stamp. If the vault holds sufficient balance, it is slashed; if not, the relayer's posted bond covers the shortfall. Thus the system guarantees that user actions are never reverted and protocol reserves cannot be drained; the only entity that ever bears economic loss is a misbehaving relayer.

5.1 Redeeming work flow

Stamps are redeemed by revealing their **secret**:

- **Secret (t):** the 16-byte random string that defines the stamp.
- **Value + binding:** the associated amt_byte, optional service_tag, and call args.

5.1.1 User side

To redeem, the user prepares a payload containing the secret (*t*), the declared credit value (*amt_byte*), and—if the stamp is service-bound—the target program identifier (*program_id*), method selector (*method_sel*), and compressed argument Poseidon (*args_hash*). These fields define the intended service call; if no binding is present, the stamp may be redeemed against any service.

The payload is transmitted anonymously through Tor (or equivalent anonymization layer) to a relayer. The client ensures locally that the same secret has not already been spent by the user.

5.1.2 Relayer side

The relayer reconstructs the commitment from the payload (using *t*, *amt_byte*, and, if present, the service binding) and derives the nullifier *N*. It checks locally that *N* has not been used in its mirror of the spent set and determines the correct tree and leaf index for the commitment in the global structure.

The relayer then submits a redemption transaction to the Stamp contract containing only the compact inputs for all stamps in the batch: *(t, amt_byte, program_id, service_tag, l0_root_id, leaf_id)*. The contract can process multiple secrets at once, deriving commitments and nullifiers for each.

Whether a relayer forwards the redemption depends on its economic incentive. Each relayer is free to set its own fee policy, reflecting capital costs, execution overhead, and risk tolerance. Clients discover and select relayers in this fee market; the protocol itself enforces only that the declared fee is covered within the redeemed credit, not what the fee must be. This keeps settlement mechanics uniform while allowing pricing to adjust dynamically through competition.

5.1.3 On-chain contract side

Upon receiving a redemption transaction, the Stamp contract deterministically recomputes the commitments and nullifiers from each submitted secret. For each secret, the program derives the commitment leaf and its corresponding nullifier, then checks the short-term Bloom filters (§6.1) to ensure that the nullifier has not already appeared within the recent slot window. Crucially, this check is made against the filter state committed before the redemption, not including the pending insertions, so that honest relayers are never flagged as double-spenders by their own submissions.

If the nullifier is absent, it is then inserted into the active filter, establishing a first-seen lock that prevents shotgun double spends during the current window. Once all submitted secrets pass this test, the program approves the batch optimistically. The associated service call is executed atomically within the same transaction, so for the user the redemption is final. For the relayer, settlement is deferred: the redeemed credit amount is advanced from the central pool into the relayer's bonded payout vault, where it remains unvested until the challenge horizon has elapsed. During this period—and for as long as the stamp remains valid (30 days)—watchers may prove non-inclusion or double-spends against the global commitment tree or the rolling nullifier tree.

If no valid challenge is raised, the balance vests and becomes withdrawable by the relayer. If a challenge succeeds, the disputed redemption is reassigned: the unvested balance is redirected to the watcher, and the relayer loses the claim.

At the same time, each redemption in the batch is appended to the relayer's rolling hash trail (§7.1.4). The Payout PDA extends

H_rel_next = Poseidon(H_rel_prev || event_leaf),

where event_leaf commits to (TAG_EVENT, l0_root_id, leaf_id, nullifier, amt_byte, bind_tag) for each consumed stamp. This ensures that all redemptions are permanently chained into the relayer's audit history, providing deterministic anchors for later watcher challenges.

Separately, all nullifiers in the batch are appended to the global rolling nullifier hash, which serves as the immutable anchor for constructing the long-term nullifier Merkle tree. This tree is maintained off-chain by relayers and watchers using the same pyramid structure described in Chapter 3, ensuring that every nullifier remains provable and auditable across the full 30-day validity horizon.

This design allows multiple redemptions to finalize instantly in a single transaction, while relayer settlement is secured through short-term Bloom-filter race protection, delayed vesting of payouts, and long-term cryptographic auditability across the full validity horizon.

5.1.4 Watcher side

External watchers provide the final layer of accountability in the redemption process. Their role is to continuously audit redemptions against the published state of the global commitment tree and the rolling nullifier tree. Challenges are not limited to a brief interval: they may be raised at any point during the full validity horizon of a stamp (up to 30 days from issuance or reshake).

Two categories of fraud can be proven on-chain:

- **Non-inclusion (commitment fraud).** The watcher provides a compact Merkle proof showing that the referenced commitment (l0_root_id, leaf_id, secret/commitment) never appeared in the claimed global root.
- **Double-spend (nullifier fraud).** The watcher proves that the same nullifier N already appeared in a prior nullifier root within the rolling nullifier tree.

To claim slashing rewards, watchers must present their proof in the context of the relayer's rolling hash trail, anchored in the Relayer PDA (§7.1.4). The PDA commits to a bounded audit window (e.g. 1–2 hours) of redemptions via start and end hashes (H_start, H_end). The watcher must validate the entire sequence of redemptions within that window and submit a proof that reconciles against both anchors. The proof includes (H_start, H_end), the computed total value of fraudulent redemptions in the interval, and the reconciled balances. On-chain, the protocol first restores the fraudulent amount to the pool from the relayer's **Payout PDA** (unvested rewards). Any remaining unvested balance in that vault is then awarded to the watcher as bounty. In addition, the relayer's **Stake PDA** (bond) is fully slashed and transferred to the watcher. This two-vault structure ensures that fraudulent redemptions are always repaid in full, prevents cherry-picking of isolated errors, and guarantees complete, window-level accountability.

The user's service action remains final—because it was executed and prepaid by the relayer—but the economic loss is shifted entirely onto the dishonest or negligent relayer, while both the pool and the user stay whole.

This structure ensures durable accountability. Even if watchers are offline for minutes or hours, they retain the ability to audit and prove fraud across the entire 30-day window. Relayers therefore operate under continuous scrutiny: their profits vest only after surviving a long challenge horizon, and any attempt to exploit gaps in monitoring results in forfeiture of earnings to the first watcher that provides proof.

5.2 Race Conditions

Race-condition protection exists primarily to defend relayers against fraudulent clients who attempt to double-spend the same stamp by broadcasting it to multiple relayers simultaneously. Honest clients only interact with a single relayer at a time and therefore never trigger this mechanism.

Duplication is prevented by the short-term nullifier filter (§6.1). Each submitted nullifier is checked against the active filter for the current slot window; the first valid submission succeeds, and all subsequent attempts within the same window are rejected deterministically on-chain.

No punitive slashing is required for honest relayers who simply lose a race; they are rejected without payout. The same rule applies in batch submissions—only the first occurrence of each nullifier is accepted, while duplicates are ignored. The result is a race-free settlement process that preserves fairness and efficiency while minimizing permanent on-chain state.

Hygiene note. An order-fixing relay (fair ordering or commit–reveal) can be used to prevent frontrunning of redemption submissions. This is optional but recommended in high-value deployments.

5.3 Payout Caps and Throttling

A relayer's profit is not immediately withdrawable: all earnings vest in a time-locked vault for 24 hours (Payout PDA), ensuring that dishonest relayers cannot extract profits and disappear before challenges are resolved. However, the redeemed credit value itself is advanced instantly by the pool when the relayer executes a service, meaning that unchecked redemptions could allow a malicious relayer to temporarily drain liquidity.

To mitigate this, the protocol enforces payout caps. At the transaction level, the total redeemed value across all Stamp calls within a single Solana transaction must not exceed a fixed maximum (protocol parameter, e.g. 0.1–1.0 SOL). This cap bounds the immediate exposure from any single user action.

At the relayer level, throughput is further throttled by stake. Each relayer maintains a bonded stake account (Stake PDA) that can be slashed in case of proven fraud. Redemption throughput is tied to this stake: the larger the bond, the higher the aggregate redemption volume the relayer can process per slot or per epoch. This ensures that any attempted abuse is economically covered by the relayer's own locked capital, while still allowing honest relayers to scale efficiently with sufficient stake (§7.2).

Together, per-transaction caps and stake-based throttling guarantee that worst-case pool outflows remain bounded, watcher challenges remain effective, and liquidity risk is matched by relayer collateral.

6. Double-Spend Prevention

Each stamp is a bearer instrument revealed by its secret. To prevent reuse, every redemption derives a **nullifier** — a one-time cryptographic marker — which must be accepted only once by the protocol. Ensuring single acceptance requires defenses across multiple horizons: a short-term guard to block "shotgun" duplicates within nearby slots, and a long-term mechanism to enforce the definitive spent set across the full 30-day validity window.

6.1 Short-term on-chain nullifier (double-buffer Bloom filter)

To prevent "shotgun" double-spends across nearby slots without storing per-redeem state, the program maintains a double-buffered Bloom filter keyed by nullifiers. This provides an O(1) check/insert path and a rolling first-seen guard spanning ~20 slots with negligible memory and compute.

6.1.1 Nullifier definition

We define N = Poseidon(tag_nullifier || t) with a fixed domain separator tag_nullifier. The nullifier is derived from the secret but does not reveal it.

6.1.2 Filter structure

Two program accounts maintain Bloom bitmaps: **F_A** (current) and **F_B** (previous). Each filter spans a fixed window of S slots. While new nullifiers are inserted into F_A, F_B continues to hold the prior S-slot segment. To check membership, the program queries the union F_A ∪ F_B, giving continuous coverage of the last 2S slots. When the slot height reaches the next S-slot boundary, rotation occurs: F_B is cleared and replaced with the old F_A, while F_A is reset to begin filling the new segment.

With S = 20 by default, this scheme ensures a rolling guard window of 20–40 slots—always covering at least 20 slots, never more than 40. Parameters are governance-tunable, and the default footprint targets ~24 bits per expected entry with $k \approx 17$ hash functions ($k \approx b \cdot \ln 2$).

Hygiene note. Relayers should use a transaction expiry shorter than the Bloom filter window (e.g. < 20 slots with the default parameters). This ensures redemptions cannot be delayed past the guard horizon, preventing a nullifier from re-entering after rotation.

6.2 Long-horizon nullifier mirror (off-chain)

Beyond the immediate Bloom-filter guard, every accepted nullifier is appended to a rolling hash chain, exactly as described for commitments (§3, §4). At fixed intervals, these rolling segments are sealed into Merkle roots and folded upward through the same five-layer pyramid (L0–L4). The current L4 root is checkpointed on-chain once per epoch, giving a durable anchor of the global spent set.

This structure serves as the authoritative record of all redeemed nullifiers within the 30-day horizon. From it, two key checks are possible:

- **Inclusion:** proving that a given nullifier was already accepted into the spent set.

- **Non-inclusion:** proving that a nullifier was *not* present at the time of redemption (e.g., to catch fraudulent activations that reference leaves never issued).

Because the same pyramid structure is used, inclusion paths and proof sizes are bounded, and state rotation follows the same ring-buffer discipline. The only difference is the data source: commitments feed the issuance accumulator, while nullifiers feed the spent-set accumulator.

7. Relayer Accounts

Each relayer is represented by two program-derived accounts:

- a **Stake PDA**, which holds bonded capital and defines throughput capacity, and
- a **Payout PDA**, which holds vesting profits, settlement flows, and the redemption audit trail.

Both accounts are derived deterministically from the relayer's public key:

- StakePDA = derive("tag_stake", relayer_pubkey)
- PayoutPDA = derive("tag_payout", relayer_pubkey)

Together, these accounts separate bonded stake from vesting rewards, while ensuring both remain fully slashable in case of fraud. Withdrawals from either account are subject to the same vesting delay (e.g. 24h), giving watchers a guaranteed horizon to contest misbehavior before funds leave the system.

7.1 Stake PDA

The Stake PDA contains the relayer's base stake, a minimum bonded amount (protocol parameter, e.g. 10–100 SOL) that must be maintained at all times. This base stake is fully slashable and guarantees that:

- **Watcher bounty guarantee.** Any proven fraud always yields at least the base stake to the first watcher who submits proof.
- **Probe deterrence.** Misbehavior or spam requires risking real capital, preventing adversaries from cheaply probing watcher activity.

Throughput control (§7.3) is derived solely from the Stake PDA balance: the larger the stake, the higher the redemption capacity.

7.2 Payout PDA

The Payout PDA holds all settlement flows: when a redemption is processed, the credit value is advanced from the central credit pool into this account. Profits remain subject to a fixed vesting delay (e.g. 24h) before withdrawal. During this period, watchers may challenge redemptions and trigger slashing.

The Payout PDA also maintains the **audit trail of redemptions**:

- Each redemption extends a rolling hash H_rel_next = Poseidon(tag_h_rel_prev || event_leaf).
- event_leaf commits to (tree_id, leaf_id, nullifier, amt_byte, bind_tag).

- (H_start, H_end) anchors define a bounded audit window (e.g. 1–2h). Watchers must validate the entire sequence between these anchors when proving fraud.
- Two checkpoint sets are retained:
    - **Anchors:** hourly (or sub-hourly) (H_start, H_end) pairs for the last 1–2 hours.
    - **Recent checkpoints:** recorded at a fixed cadence (e.g. per slot), each paired with the Payout PDA balance.

This dual system enables both immediate challenges and bounded re-audits while preventing relayers from flooding checkpoints to overwhelm auditors.

7.3 Throughput Control

A relayer's redemption capacity is strictly proportional to its Stake PDA balance. The protocol defines a base unit U (e.g. 1 SOL). Capacity is computed as:

- max_redeems = floor(stake_balance / U)

Two regimes follow:

- **Normal regime (capacity ≥ 1).** The relayer may process up to max_redeems redemptions per slot.
- **Low-capacity regime (capacity < 1).** The relayer may redeem at most once every Δ = ceil(1 / capacity) slots.

This spacing rule prevents under-collateralized relayers from spamming consecutive redemptions, while still keeping the system open to new entrants with small stakes.

Example. With U = 1 SOL:

- A relayer with 50 SOL bonded has capacity = 50. It may process up to 50 redemptions per slot.
- A relayer with only 0.5 SOL bonded has capacity = 0.5. It may redeem only once every Δ = ceil(1/0.5) = 2 slots.

Each Stake PDA maintains two counters to enforce these rules:

- last_slot — the slot of the most recent redemption.
- redeem_count — the number of redemptions already executed in that slot.

On submission, the contract compares the request against the relayer's capacity, last_slot, and redeem_count; if the quota for the slot is already filled in the normal regime, or if the required spacing has not elapsed in the low-capacity regime, the redemption is rejected, otherwise it is accepted and the counters are updated accordingly.

This mechanism ensures that:

- **Throughput is always collateralized.** No relayer can process more value per slot than its stake supports.
- **Scale requires capital.** Relayers who bond more stake scale linearly; those who withdraw reduce capacity immediately.
- **Safety for small players.** Even with small stakes, relayers remain eligible, but throttled to safe redemption rates.

7.4 Withdrawal

Withdrawals from both the Stake PDA and the Payout PDA are governed by a single-note mechanism with a fixed vesting delay (protocol parameter, e.g. 24 hours). When a relayer requests a withdrawal of amount x, the program records a note (x, unlock_slot) where unlock_slot = current_slot + vesting_delay. Only one note may exist per account at any time; submitting a new request overwrites the previous note and resets the timer.

Once the unlock slot is reached, the relayer may withdraw up to x from the account's current balance. If the balance has been reduced by slashing in the meantime, the relayer can only withdraw the remainder; the difference is forfeited.

This model ensures three things:

- **Simplicity.** At most one withdrawal is pending per account.
- **Predictability.** Vesting delay is fixed and transparent.
- **Safety.** Watchers are guaranteed a full challenge horizon before any capital—whether bonded stake or unvested payouts—can leave the system.

7.5 Settlement

When a redemption is executed:

- The relayer pays Solana transaction fees from its external wallet.
- The redeemed credit value is transferred from the central credit pool into the Payout PDA. These funds follow the standard withdrawal rules defined in §7.4: they remain unvested until a withdrawal request matures after the vesting delay (e.g. 24h).

1. If fraud is proven during this period, the unvested Payout PDA balance is first used to restore the pool, and any remainder goes to the successful watcher.
2. Separately, the entire Stake PDA is slashed and transferred to the watcher.

Each redemption extends the audit trail stored in the Payout PDA (§7.2), ensuring that settlement and accountability are tightly linked. This two-vault design guarantees that fraudulent redemptions are always repaid, that watchers are rewarded with meaningful bounties, and that end-user actions remain final.

7.6 Security Properties

- **Bounded exposure.** A relayer cannot process more value per slot than its bonded stake supports.
- **Delayed profits.** Settlement flows vest only after the challenge horizon, ensuring watchers can contest fraud.
- **Dedicated risk pools.** The Stake PDA anchors accountability with bonded capital, while the Payout PDA contains settlement flows and the audit trail.
- **Robust incentives.** Watchers are always compensated: the Payout PDA restores the pool first, and the Stake PDA guarantees a bounty.

This separation ensures that scale requires stake, profits remain slashable, and any misbehavior burns capital across both accounts.

8. Infrastructure Economics

Operating the Stamp Protocol requires maintaining off-chain mirrors of three data sets:

- **Commitments** (credits issued, §3),
- **Nullifiers** (credits spent, §6), and
- **Redemptions** (per-relayer event trails, §7.1.4).

Relayers rely on these mirrors to pre-filter duplicates, reconstruct Merkle paths, and extend their PDA audit chains. Watchers rely on them to audit redemptions, validate rolling-hash anchors, and generate on-chain fraud proofs.

Because all three datasets share the same pyramid/Merkle structure, the infrastructure requirements for relayers and watchers overlap almost completely.

8.1 Aggregator Infrastructure

Aggregators are responsible for sealing commitments into Merkle roots and publishing them on-chain. Unlike relayers and watchers, they do not need nullifier or redemption data—only the issuance stream. Their workload is characterized by:

- **Commitment mirrors** — maintain a complete ordered copy of commitments and the rolling issuance hash.
- **Proof generation** — run zero-knowledge circuits to prove that sealed ranges extend the rolling hash and map correctly into Merkle roots. This is the most resource-intensive task, requiring sustained proving capacity (CPU or GPU).
- **On-chain submissions** — monitor unsealed ranges and race to publish valid roots; the first valid proof per range is paid a sealing fee.

Protocol fees:

- A fixed reward per entry sealed into the commitment tree (e.g. 500 lamports per commitment).
- A fixed reward per entry sealed into the nullifier tree (e.g. 500 lamports per nullifier).

Because the issuance dataset is smaller than the combined commitment+nullifier+redemption mirrors, aggregator storage and bandwidth demands are light. The main differentiator is proving throughput: competitive aggregators will invest in fast proving hardware or cloud farms.

In practice, many operators will bundle aggregation with relayer or watcher functions, since the mirrors largely overlap and sealing only adds proving overhead.

8.2 Relayer infrastructure

Relayers must maintain up-to-date mirrors of the commitment and nullifier pyramids (§3, §6). These mirrors allow them to:

- Pre-filter repeats locally, avoiding wasted transaction fees.
- Reconstruct Merkle paths for commitments and nullifiers when forwarding activations or redemptions.
- Provide proof services to clients, who may not maintain their own local mirrors.

8.3 Watcher infrastructure

Watchers maintain the same mirrors as relayers—commitments (§3), nullifiers (§6), and additionally per-relayer redemption trails (§7.1.4). These datasets allow them to:

- Continuously audit redemptions against the published roots and PDA anchors.
- Detect inconsistencies, non-inclusions, or double-spends.
- Prepare Merkle and rolling-hash proofs for on-chain challenges.

In addition, watchers must process every redemption proof. Each submitted proof is compact (~200 bytes) but must be checked on arrival. Watchers may either persist these proofs alongside the redemption trail or refetch them when validating an audit window (e.g. 1–2h). This ensures that when a fraud proof is prepared, the watcher can reconstruct the full sequence of redemptions between anchors without gaps.

8.4 Shared costs

All three roles—aggregators, relayers, and watchers—mirror much of the same data. The difference lies mainly in *what they prove*:

- **Aggregators** prove issuance ranges extend the rolling hash.
- **Relayers** pre-filter nullifiers and extend redemption trails.
- **Watchers** audit both, replaying proofs and generating fraud challenges.

Because the mirrored datasets (commitments, nullifiers, redemption trails) are shared, the baseline infrastructure is nearly identical across roles:

- **Storage.** At ~10k redemptions/sec, the system produces ~864M nullifiers/day. At 16 B each, that's ~13.8 GB/day and ~415 GB/30 days per stream. Adding commitments roughly doubles this, giving ~27.6 GB/day and ~830 GB/30 days. With indexes/caches, round to ~1 TB working set per mirror. Per-redemption proofs are ~200 B each: ~2 MB/sec ≈ 7.2 GB/hour, small compared to the commitments+nullifiers stream.

- **CPU.** Poseidon hashing, Bloom filters, and Merkle lookups remain well within commodity hardware capacity. Aggregators additionally run proving circuits, which may benefit from GPU/accelerator support.
- **Bandwidth.** Steady sync with on-chain roots plus peer-to-peer exchange of Merkle paths between mirrors.

Because of this overlap, most operators are expected to run multiple roles together on the same cluster. The incremental cost of adding one function on top of another is minimal; the only major extra requirement is proving throughput for competitive aggregation.

Optimization. Watchers can reduce footprint further by leaning on relayer infrastructure: e.g., fetching Merkle paths from relayer mirrors while only keeping a 24-hour local window (~40 GB). This lowers requirements to a single mid-range server, yet still allows full fraud detection and proof submission within the validity horizon.

8.5 Cost model

A minimal deployment consists of:

- One archival node storing the full 30-day mirror (~1 TB).
- Hot cache layers (RAM + NVMe) for the last ~24h (~40 GB) to accelerate queries.

At current cloud pricing, a dedicated server with 2 TB NVMe, 16 cores, and 64 GB RAM is <$500/month. This is well within expected operator revenue:

- **Relayers** earn margin on every redemption/activation they forward.
- **Watchers** receive slashing rewards whenever they prove fraud.

Together, these incentives cover infrastructure costs comfortably and create a strong economic motive for independent operators to maintain resilient, geographically distributed infrastructure.

8.6 ROI Gradient for Relayers

Relayer profitability scales inversely with redemption size.

- **Micro-redemptions:** At denominations close to the Solana fee floor (≈0.000005–0.000010 SOL), the margin over fees may only be a few thousandths of a SOL in absolute terms, but it represents double-digit percentage ROI on the capital locked in the Payout PDA for 24 h. For example, if a message credit pays 0.0000075 SOL, the relayer covers a 0.000005 SOL transaction fee and clears 0.0000025 SOL. Though small, this is ~50% return on the 0.000005 SOL advanced for that redemption.
- **Larger redemptions:** At 0.1–1 SOL credits, the absolute margin grows (hundreds to thousands of times larger), but the ROI shrinks to the 0.1–1% daily range, closer to validator-style economics.

This gradient creates a natural equilibrium:

- Small credits incentivize relayers to service high-throughput, low-value applications like messaging, micro-APIs, and pay-per-query services.
- Large credits are still viable but mainly routed through reshakes, serving as liquidity consolidation rather than high-ROI work.

The design therefore tilts economics toward micropayments, ensuring relayers are eager to process small, frequent spends while the protocol maintains bounded exposure and secure settlement.

8.7 Fee Structure Overview

For clarity, the protocol's fee mechanics are:

- Aggregator sealing fees: fixed protocol fee per entry sealed into commitments and nullifier trees (e.g. 500 lamports each).
- Relayer reshake fees: residual after issuing fresh commitments, drawn from the difference between input and output credit values.
- Relayer redemption margin: excess after covering Solana fees and execution costs.
- Watcher rewards: funded by slashed relayer stakes and unvested payouts when fraud is proven.

This balance ensures each role is incentivized: aggregators are paid for liveness, relayers profit from servicing usage, and watchers profit from catching fraud.

9. Privacy Properties

The central goal of the Stamp Protocol is **unlinkability**: no observer should be able to connect a redeemed credit to the account that originally funded it. To evaluate this claim, we summarize the privacy properties against different adversaries, highlight the role of reshaking, and clarify the boundaries of protection.

9.1 Threat model

We assume adversaries may control any single role (issuer, aggregator, relayer, or watcher) or even collude across them. The protocol must still guarantee that:

- Funding accounts cannot be linked to later redemptions.
- Transfers of credits (reshaking or secret sharing) cannot be traced.
- Users can spend without maintaining long-lived accounts.

9.2 Guarantees

- **Issuance privacy.** Commitments hide both secrets and values; the on-chain program only verifies zero-knowledge proofs. Even an issuer cannot tell which credits belong to which user once they have been reshaken.
- **Aggregation privacy.** Aggregators observe only blinded commitments and produce Merkle roots; they cannot link roots back to individual payers.
- **Reshaking unlinkability.** At issuance, commitments are still tied to a funding account. If their raw secrets were shared at this stage, redemption would reveal a direct link to the payer. Reshaking breaks this link: the user proves in zero knowledge that they can consume an old commitment included under the aggregator's published root and reissue its value into a fresh one. The proof reveals only the nullifier, never the secret or the original commitment, so observers cannot connect the new credit to any payer. From this point on, the new secret can be passed around safely like a bearer token.
- **Redemption privacy.** Relayers see only short secrets and value bytes. Because reshaking severs the link between issuance and redemption, the relayer cannot distinguish whether a secret originated from the original buyer or was transferred through others.
- **Watcher privacy.** Watchers audit correctness but see only nullifiers and Merkle proofs. These reveal that a credit was spent, but not who funded or transferred it.

9.3 Boundaries

Privacy is not absolute. The protocol does not prevent:

- **Network-layer correlation.** Timing analysis between client and relayer can leak information. Onion routing or equivalent anonymization is recommended to reduce this risk.

- **Value-class inference.** Denomination information is not revealed at issuance: the zero-knowledge proof only exposes the number of stamps and their total prepaid value. Leakage occurs only at redemption, when the one-byte value class is disclosed. Users can mitigate inference by reshaking into uniform denominations or inserting decoys, so that individual redemptions do not betray original funding patterns.
- **Service-level metadata leakage.** The fact that a specific service was invoked is observable to the relayer providing it.

10. Applications

The Stamp Protocol enables unlinkable prepaid usage credits. While its construction is general, several concrete applications highlight why such a primitive is needed:

10.1 Privacy-Preserving Messaging

Consider a secure messenger that charges a small fee per message to prevent spam and denial-of-service attacks. Without unlinkability, the payer of credits could always be linked to the message sender, undermining privacy guarantees.

With S.T.A.M.P.:

- A user purchases a bundle of stamps once, funding the pool.
- These stamps can be redeemed by anyone (the user, or gifted to a contact).
- When a message is sent, the relay node only sees a valid stamp spend, not the identity of the original payer.
- This separates **payment identity** from **communication identity**, eliminating metadata leakage between funding and usage.

10.2 Metered Services (APIs, Cloud, IoT)

Many services charge per request—whether web APIs, cloud microservices, or IoT bandwidth—but today these rely on API keys or account-based billing. This ties every usage event back to a payer and leaks metadata about who consumed what.

With S.T.A.M.P.:

- A user prepays once and receives a bundle of stamp secrets.
- Each request carries one stamp secret as an ephemeral "access credit."
- The backend validates the secret through a relayer, who executes the call and redeems the credit.
- The service is paid fairly, but cannot link usage back to the original funding account.

10.2.1 High-throughput batching

For popular services, redeeming each stamp individually may generate high transaction load. To handle this, providers can:

- Batch many secrets into a single Solana transaction (≈40–50 per tx).
- Or reshake many small stamps into one larger leaf, consolidating value while unlinking individual requests.

Both paths preserve unlinkability; batching optimizes throughput, reshaking optimizes liquidity and hides request granularity.

10.2.2 Beyond APIs

The same model extends naturally to machine-to-machine environments like cloud microservices or IoT. Devices can carry prepaid stamps without accounts, spending them per function call or per byte of bandwidth. Because stamps are transferable, credits can even move between devices directly, enabling prepaid resource usage without centralized billing systems.

10.3 Anti-Abuse and Fair-Use Environments

Open platforms (forums, voting systems, collaborative tools) often suffer from spam and Sybil attacks. Traditional CAPTCHAs or centralized tokens are weak, centralizing, or annoying.

With S.T.A.M.P.:

- Each action (post, vote, submission) can cost a tiny prepaid stamp.
- Since stamps are unlinkable, no long-term identity is needed.
- Abuse is disincentivized economically, while honest participation remains frictionless.

11. Conclusion

The Stamp Protocol provides a practical design for unlinkable prepaid service credits. By combining cryptographic commitments, Merkle tree batching, zero-knowledge proofs, and on-chain slashing, it achieves privacy and scalability without relying on trusted intermediaries. Like postage stamps for the digital age, S.T.A.M.P. enables private, decentralized, and efficient payments for network services.

Appendix A: Cryptographic Conventions

All Poseidon and SHA256 hashes in S.T.A.M.P. use fixed domain-separation tags to prevent re-use collisions. Each tag is a single-byte constant prepended to the hash input. Truncation is low-end 128-bit unless otherwise specified.

Pseudorandom Function choice (PRF):

- PRF_t(x) = Poseidon(t || x) over BN254 Fr.
- All tags are **single bytes** shown as hex; payloads are byte strings unless noted.

Tag	Name	Purpose / Construction
10	**TAG_CT**	Token commitment: C_t = Poseidon(0x10 || t)
11	**TAG_CV**	Value/service commitment: C_v = Poseidon(0x11 || amt_byte || (svc_tag?))
12	**TAG_LEAF**	Commitment leaf: L = Poseidon(0x12 || C_t || C_v)
13	**TAG_EVENT**	Redemption event leaf: event_leaf = Poseidon(0x13 || tree_id || leaf_id || nullifier || amt_byte || bind_tag)
14	**TAG_H_ISS**	Rolling issuance hash: H_iss_next = Poseidon(0x14 || H_iss_prev || commitments_batch)
15	**TAG_H_NULL**	Rolling nullifier hash: H_null_next = Poseidon(0x15 || H_null_prev || nullifier)
16	**TAG_H_REL**	Relayer audit hash: H_rel_next = Poseidon(0x16 || H_rel_prev || event_leaf)
17	**TAG_SVC**	Service binding tag: svc_tag = Trunc128(Poseidon(0x17 || program_id || method_sel || args_hash))
18	**TAG_MERKLE_NODE8**	Arity-8 Merkle internal node: node = Poseidon(0x18 || child0 || child1 || child2 || child3 || child4 || child5 || child6 || child7)
19	**TAG_TREE_ID**	Tree identifier derivation: tree_id = Trunc128(Poseidon(0x19 || domain))
1A	**TAG_ROOT_META**	Root metadata binding: root_meta = Trunc128(Poseidon(0x1A || epoch || layer || index))
20	**TAG_NULL**	Nullifier: N = Trunc128(Poseidon(0x20 || t))

30	**TAG_HANDLE**	Leaf handle (16 B): handle = Trunc128(Poseidon(0x30 || L || root_id))
60	**TAG_ROOT_ID**	Root identifier (SHA side): root_id = Trunc128(SHA256(0x60 || poseidon_root || meta))

Notes

- All concatenations are unambiguous (fixed-length encodings: t = 16 B, amt_byte=1 B, hashes=32 B before truncation).
- Truncation is **low-end 128-bit** unless otherwise specified.
- Reserve 0x80–0xFF for future extensions (refunds, payer binds, fee caps, etc.).
- Poseidon is used for all commitments and Merkle paths because it is SNARK-efficient, while SHA-256 is used for aggregator receipts and slashing proofs because it is standard, hardware-accelerated, and widely verifiable outside ZK.

Hygiene note: Never confuse root_id with the canonical root. The program verifies against the full 32-byte Poseidon root; root_id is just a short identifier for efficient indexing and references.

Security hygiene

- Never reuse a tag in a different context.
- Do not allow variable-length fields without a length prefix—here we use only fixed-length fields.
- Document these constants in the spec and freeze them at **version 1** (put protocol version in extras high nibble if desired).
- Domain separation ensures that all values derived from the same secret remain independent, non-colliding, and unlinkable: even if t is reused, commitments, nullifiers, and randomizers evaluate to distinct pseudorandom outputs, preserving both soundness and privacy.

All Poseidon and SHA256 hashes in S.T.A.M.P. use fixed domain-separation tags to prevent re-use collisions across different contexts. Each tag is a single byte constant concatenated into the hash input. Table 1 lists the assigned tags.

Appendix B: Credit Value Classes

Each stamp represents a prepaid unit of service credit. To keep values both human-friendly and privacy-preserving, S.T.A.M.P. uses a discrete set of denominations encoded in 6 bits.

Cls	Value	Note
0	0.000005	**Burn-only** (≈ base fee)
1	0.000006	redeemable micro
2	0.000007	redeemable micro
3	0.000008	redeemable micro
4	0.000009	redeemable micro
5	0.000010	redeemable micro
6	0.0000125	
7	0.000015	
8	0.0000175	
9	0.000020	
10	0.000025	
11	0.000030	
12	0.000040	
13	0.000050	
14	0.000075	
15	0.000100	
16	0.000250	

17	0.000500	
18	0.001000	
19	0.002000	
20	0.003000	
21	0.005000	
22	0.010000	
23	0.020000	
24	0.050000	
25	0.100000	
26	0.200000	
27	0.500000	
28	1.000000	per-tx cap zone
29	2.000000	reshake/split
30	10.000000	reshake/split
31	100.000000	reshake/split

Appendix C: Protocol Parameters

Name	Symbol	Default	Scope	Change Policy	Notes
L0 arity	A	8	On-chain param	Epoch-boundary	Arity of Merkle nodes (Poseidon).
L0 depth	D0	4	On-chain param	Epoch-boundary	Leaves per slot = A^D0 (4,096 @ A=8).
L1 cadence (slots→minute)	—	~150 slots	Fixed	—	One L1 per minute.
L2 width (minutes→hour)	—	60	Fixed	—	One L2 per hour.
L3 width (hours→day)	—	24	Fixed	—	One L3 per day.
L4 width (days→window)	—	30	Fixed	—	~30-day retention.
Retention horizon	T_ret	30 days	Fixed	—	Roots kept/valid for challenges.
Bloom window (per segment)	S	20 slots	On-chain param	Immediate	Double-buffered; coverage 20–40 slots.
Bloom bits / expected entry	b	24	On-chain param	Immediate	Guides filter size.
Bloom hash count	k	⌊b·ln2⌉ (≈17)	Derived	Immediate	Standard optimum.
Tx expiry (relayer hint)	—	< S slots	Off-chain policy	—	Hygiene: avoid expiry beyond Bloom.
Nullifier size	—	16 bytes	Fixed	—	`Trunc128(Poseidon(TAG_NULL

Secret size	—	16 bytes	Fixed	—	Stamp secret t.
Denomination classes	—	16 (4-bit)	On-chain param	Governance	Table of amt_class→value.
Vesting delay	T_vest	24 h	On-chain param	Governance	Applies to Stake & Payout PDAs.
Min base stake	S_min	10–100 SOL	On-chain param	Governance	Must be maintained in Stake PDA.
Throughput base unit	U	1 SOL	On-chain param	Governance	Capacity = ⌊stake/U⌋ per slot.
Per-tx payout cap	C_tx	0.1–1.0 SOL	On-chain param	Governance	Max total redeemed per Solana tx.
Audit window (relayer trail)	T_audit	1–2 h	On-chain param	Governance	(H_start, H_end) window.
Trail checkpoints (recent)	N_ckpt	100	On-chain param	Governance	Stored with Payout PDA balance.
Anchor cadence	—	hourly / sub-hourly	On-chain param	Governance	Pairs of (H_start, H_end).
Aggregator sealing fee	F_seal	—	On-chain param	Governance	Paid per sealed range/L1.
Challenge bounty split	—	Payout first, Stake full	Fixed	—	Payout redeems pool, remainder + full Stake → watcher.
Hash functions	—	Poseidon (BN254), SHA-256	Fixed	—	Poseidon in-circuit/trees; SHA for optional receipts/IDs.
Domain tags	—	See Appendix A	Fixed	Versioned	Freeze per protocol version.

Change policy notes

- *Epoch-boundary*: takes effect only from the next L4 epoch; old roots remain valid until expiry.
- *Governance*: via config account; publish config_hash and bind new L0 roots to it.