Community Credits
A Private, Closed-Loop Payment System for the SimpleX Network

Evgeny Poberezkin ep@simplex.chat Alain Brenzikofer alain@simplex.chat

July 7, 2026

Abstract

SimpleX is a messaging network with no user profile identifiers and strong metadata privacy. It leaves an open question – how a decentralized, censorship-resistant network can pay the operators and developers it depends on: a general-purpose payment system would reintroduce the identification the network avoids. We present *Community Credits*, a closed-loop payment system that funds network capacity while preserving user privacy. Credits are prepaid, non-transferable, expiring claims on capacity, backed one-for-one by a stablecoin held in a non-custodial smart contract. They are bought openly, assigned to communities in private, and redeemed in private by those communities with a known, registered set of operators, who are paid under a contract-enforced revenue-sharing agreement. We set out the design objectives and threat model, a note-based zero-knowledge protocol, the regulatory grounding for the closed-loop structure, and a method for reading blockchain state without revealing the querier. Each redemption is visible as an on-chain event, but the design conceals its attributes, the amount, which operator is paid, and the expiration cohort (beyond a coarse bracket set by the input note's epoch), and keeps redemptions unlinkable to one another and to the purchase and assignment that preceded them, so an operator's earnings become visible only in aggregate at withdrawal. We state the residual limits and the parameters left open.

Contents

1 The SimpleX Network

SimpleX is a privacy- and security-focused messaging network with no user profile identifiers: no accounts, phone numbers, usernames, or random numbers are assigned to users by the network. With no identifier to observe, servers cannot build a social graph, and users keep full control over their identity, contacts, and groups. The application has over two million downloads and around 400,000 monthly active users, with close to a thousand servers run independently by the community.

The network delivers data packets over the public Internet through nodes (SMP relays) that accept, temporarily store, and forward packets without holding user accounts. Messaging uses unidirectional queues addressed by per-connection credentials rather than user identity [10]; two-hop onion routing ("private routing") protects the sender's network address from the recipient's relay, and traffic is padded into fixed-size blocks to frustrate correlation. Content is end-to-end encrypted between clients with a continuous post-quantum double ratchet [12], independent of the relays that deliver it. Files are sent by a separate protocol, XFTP [11], which splits an encrypted

file into chunks spread across relays. On top of this transport, *channels* [13] provide stateful one-to-many delivery. Channel identity and content are controlled by keys the channel owner holds, so no operator can seize or alter them. The network overview [7] gives a fuller account.

2 Motivation

A sustainable, user-controlled network must pay its operators, and it cannot do so through a general-purpose payment system without undermining the metadata privacy SimpleX is designed to provide. Three points follow, taken up in turn: the network's metadata privacy depends on independent, non-colluding operators; the registry that lists them must resist censorship; and operators and developers must be paid for their work.

2.1 Collusion Resistance

Metadata privacy in SimpleX assumes that the relays handling a user's traffic do not collude. The relevant threat is traffic correlation, not a Sybil attack. A Sybil attack forges many identities to seize control of the network itself. That takes a large share of the nodes: a majority under proof-of-work, or at least a third of a Byzantine-fault-tolerant system, which needs two-thirds honest [5, 6]. Deanonymization needs far less: an adversary who operates relays at both ends of users' connection path can link users' connections by correlating timing and volume, using only a small fraction of the relays. Open-participation networks such as Tor[1] are exposed here: participation is pseudonymous, anyone may run a relay, and a user has no assurance about who operates the relays on their connection path, so operating a small fraction of relays already correlates a meaningful fraction of streams [3, 4]. Economic barriers such as deposits or staking raise the price of running relays at scale [2], but none of this puts a small network fraction beyond a well-funded adversary [1].

SimpleX chooses a pragmatic countermeasure: beyond making collusion technically hard, it ensures that the relays a client uses are run by independent parties who have also made contractual commitments to users.

The SimpleX app by default will use the registry of whitelisted operators who have identified themselves (through domain ownership or another process) and signed an Operator Deed that commits them to a network-wide privacy policy, including a commitment not to share any users' data or metadata with third parties except under compulsory legal process, and to delete the payout-note openings and redemption records that tie their redemptions to communities once each cohort settles. The app ensures that sending, proxying, and receiving relays are operated by independent organizations that host their servers in different datacenters. Such diversity of operators reduces the risk of collusion, and also improves delivery reliability.

2.2 Censorship and Coercion Resistance

SimpleX aims to promote freedom of expression and freedom of assembly in the digital domain. Not every jurisdiction (or company) upholds these civil rights, which is why SimpleX cannot stop at private communication alone. SimpleX also aims to stay accessible at all times and in all places where people want to use it.

The operator whitelist introduced in the previous section is only effective if the registry providing this whitelist as the single source of truth is protected against censorship and coercion. Such protections cannot be provided by any single trusted party responsible for the registry's availability

[1] Tor is, in fact, permissioned (a small set of directory authorities controls which relays appear in its consensus)

and integrity. Instead, SimpleX uses smart contracts on a public, permissionless ledger: the ledger guarantees the availability and immutability of a given version of the operator whitelist, while a smart contract enforces the rules for updating it, allowing changes only by the parties the contract defines. The SimpleX app can then read and verify the latest version of the whitelist from the smart contract via onion-routed queries to indexer nodes (Section 6.5), so that no single party sees both the queried state and the querier's identity.

Definition 1. ***SimpleX Operator Registry Contract (SORC):*** *A smart contract that stores a whitelist of operators and their servers, including metadata about their identification, TLS certificates (or other means of authentication), and infrastructure location. The SORC is solely controlled by the SimpleX Network Consortium; full governance is deferred to future work.*

2.3 Economic Viability

For a decentralized SimpleX network to be sustainable, operators must be paid for the infrastructure they provide and developers for the software they build. Community Credits (Section 6) provide a privacy-preserving, censorship-resistant way to make those payments.

3 Channels and the Economic Model

Channels are where the network's scale, cost, and commercial opportunity concentrate. Private one-to-one and small-group messaging, the network's original purpose, stays free; the demand that strains infrastructure comes from large channels and communities broadcasting to wide audiences. The owners of those channels depend on infrastructure that cannot be taken from them, and they are the customers of this economic system.

Channel traffic follows a power-law distribution, as it does across networked media: on large video platforms the top 10% of videos account for roughly 79% of views [8], and on the open web roughly a hundred domains account for about a third of all traffic [9]. The large channels that generate most of the load also have the resources and the incentive to pay for reliable, censorship-resistant infrastructure. Revenue from them keeps direct messaging and small groups free for everyone else, so the network stays sustainable without the free-tier degradation common to ad-funded and subscription platforms.

The economic system prices access to network capacity and routes payment from channel owners (and, through donations, their audiences) to the Network Operators that serve them, with a share retained for the network's development and governance. Which specific service a payment buys, whether a memorable name, file storage, message delivery, or subscriber capacity, is a separate service-model question out of scope of this paper; Community Credits are the payment system.

4 Regulatory Considerations

Regulatory constraints shaped this design rather than following from it. General-purpose payment systems require know-your-customer (KYC) identification, which would destroy the user privacy the network provides. Community Credits are therefore deliberately closed-loop, a structure that preserves privacy and fits established regulatory exemptions.

Community Credits are closed-loop, non-transferable, expiring prepaid access to network capacity, redeemable only with a known, published list of Network Operators, not a tradable token, a security, or a general-purpose payment instrument. User stablecoin is held by an autonomous,

non-custodial smart contract: SimpleX does not take custody of user funds or keys, issue stablecoin, or exchange between fiat and crypto. In the United States the system is structured to fit the closed-loop prepaid-access exemption [34]: every payee is identified to users in advance through the public operator registry, and per-user purchases are limited to no more than $2,000 per day, mediated in the user interface in the same way prepaid telecom cards apply daily limits. In the United Kingdom it is structured to fit the limited-network exclusion [35]: an admission-controlled list of Network Operators bound by a signed Operator Deed, redeemable only through the SimpleX interface rather than an open marketplace. The analysis relies on the recipients of funds being identified and confined to a known list, not on the payees being regulated, while the design conceals the payer, the amount, the paying operator, and the payment's link to a community.

5 Design Objectives and Threat Model

Community Credits are a payment system: the design defines how value moves through the network. The objectives below, and the privacy goals in particular, shaped the design described in the following sections; Figure 1 (Section 6) sketches the resulting credit lifecycle from purchase to withdrawal.

5.1 Functional and Economic Objectives

- **Closed-loop utility credits.** Credits are prepaid access to network capacity: non-transferable between users, redeemable only with registered Network Operators, and expiring after a fixed lifetime.
- **Fixed-denomination credits, with private assignment and change.** Credits are issued in a small set of fixed denominations, so the public purchase amount is low-cardinality; the assigned and change amounts are private, constrained only by the minimum-spend floor M. A purchaser can assign all or part of a credit to a community and receive the remainder back as change. Only the community a credit is assigned to can redeem it, and a credit cannot be re-assigned or transferred once assigned.
- **Solvency by construction.** The face value of all recorded credits never exceeds the stablecoin actually deposited; every credit is fully backed at all times.
- **Expiring liabilities, reclaimed only in aggregate.** Value left unredeemed when a credit expires is reclaimable by the network only in aggregate, after expiry and on a delay, never for an individual credit.
- **Zero-trust settlement.** Stablecoin remains locked in the contract through purchase, assignment, and redemption. Operators accrue on-chain claims as credits are redeemed and withdraw later; the contract applies the revenue split at withdrawal; its ratio is a consortium-governed parameter.
- **Mobile-provable redemption.** The zero-knowledge proof a community produces to redeem a credit must be generated on a mid-range mobile phone within approximately one second.

5.2 Privacy Goals

There are three transaction steps (purchase, assignment, redemption), followed by operator withdrawal. Value is visible where it enters and leaves the pool and concealed in between:

- **Purchase is visible.** The amount paid for a credit is recorded on-chain. We accept this visibility.
- **Assignment is private.** Assigning a credit to a community does not reveal the amount assigned or which community received it, and cannot be linked to the purchase it came from. This is the step that converts a visible purchase into a private contribution.
- **Redemption is attribute-hiding and unlinkable.** An individual redemption is visible as an event but reveals no operator, amount, or expiration cohort (beyond the epoch bracket), and cannot be linked to another redemption or to the purchase and assignment behind it; the value is held in a sealed payout note. Redemptions cannot be linked to one another, to an assignment, or to a purchase, and the value an operator settles later at withdrawal cannot be tied back to any specific redemption.
- **Only withdrawals are public.** What becomes visible is each operator's *aggregate* earnings, disclosed at withdrawal on the operator's own schedule and on a delay; the individual redemptions that produced them stay hidden.

The overriding requirement is unlinkability: no party may connect a purchase, the community it was assigned to, and the redemptions that follow. The motivating threat is concrete. A community's chosen servers are observable, and a community may publicly appeal for funding; if payments to its servers were individually visible, a donation could be matched to a burst of payments and tied back to the community and the donor. Concealing redemptions as events, and revealing only aggregate operator earnings, removes that correlation.

5.3 Threat Model

Assumptions.

- SimpleX provides metadata-private membership and messaging, and the onion transport by which clients reach a chain-access node; the credits system must add no deanonymization channel SimpleX otherwise prevents.
- Cryptographic primitives are secure and the circuits agree with the on-chain hash, so proofs are sound; soundness further assumes at least one honest participant in each per-circuit Groth16 setup ceremony and in the folding wrapper's setup, so no toxic waste is retained. The chain and its events are public and permanent.
- Operators who may withdraw are registered and bound by the Operator Deed, which they honor: in particular, they do not collude. Their identities are public; concealing *which* operator a redemption pays is a goal, not a contradiction of this.
- The smart contract holds all the funds and enforces custody of the principal: no party, operators included, can move or seize the locked stablecoin, and no one can take the value a user can redeem. What an operator ultimately receives is the consortium-governed revenue split applied at withdrawal (see below). This is the custody and solvency the objectives above require.
- The backing stablecoin's issuer does not freeze or blacklist the contract's holdings. USDT and USDC are centrally pausable, so this is a trust assumption; a decentralized, collateral-backed stablecoin such as DAI reduces it, though not entirely.

- Enough independent, non-colluding users and communities use the system, and each operator serves many of them. The privacy guarantees depend on this: they hold when volume is high and weaken toward individual amounts when an operator has few users.
- The purchaser's web dApp and wallet have integrity: they generate the spending key sk and the note randomness honestly client-side and do not leak them.
- The SimpleX Network Consortium governs two parameters the contract applies: the revenue-split ratio taken at withdrawal, and the operator registry (admission, removal, and freezing). It is trusted for these. Setting the split adversarially would shrink what already-accrued but un-withdrawn payout notes ultimately pay their operators, and removing an operator from the registry before a cohort's finalization window closes forfeits that operator's un-withdrawn cohort notes to the treasury; neither power can reach a user's redeemable balance. We treat this as a trusted-governance assumption and defer the governance design to future work.

On-chain observer (primary adversary). Reads all on-chain activity and all network traffic reaching the chain, persistently and retroactively, and may itself act as a purchaser, community, or operator.

It *can* see each purchase (the purchaser's wallet address, the amount, and the expiration bucket), the submitter address on each assignment and redemption, and that redemptions and withdrawals occur and when; the on-chain aggregates it additionally learns are listed under accepted disclosures below.

It *cannot* achieve any linkage the privacy goals above forbid: it cannot tie a purchase to the community it funds or to later redemptions, learn which community a credit was assigned to or enumerate a community's members, or recover an individual redemption's amount, which operator is paid, or its cohort, beyond the coarse epoch bracket noted among the accepted disclosures below.

Network Operator (honest-but-curious or coerced). In practice one operator plays several roles for the communities it serves: it runs their SMP relays (so it already knows those communities out-of-band), usually the indexer that answers their chain queries, and often the submitter that relays their transactions.

It *can* learn, out-of-band, the amount and paying community of each redemption it serves, and its own total turnover. As its communities' indexer it can link a `getPath` for an assignment output to the later redemption by timing (Section 6.3.9). As submitter it can time, withhold, or reorder the transactions it relays.

It *cannot* learn a community's membership, decompose another operator's activity, be forced to reveal its customer set on-chain (there is no per-community on-chain trail), or recover the value, operator, or cohort (beyond the epoch bracket) of redemptions it does *not* serve. The a_{sub} binding (Section 6.3.4) keeps it from taking another submitter's cashback.

Independent indexer or submitter (the weaker case). An indexer or submitter run by a party that does *not* serve the community is weaker than the operator above.

It *can* match a `getPath` to the later redemption by timing, but only as two pseudonymous events, since it does not know which community is behind them; a submitter can also withhold or delay what it relays.

It *cannot* learn the querier's identity, because onion routing keeps who-asks separate from what-is-asked (Section 6.5); and a relayed payload reveals no amount or operator, no cohort beyond the epoch bracket, and its outputs cannot be changed.

Accepted disclosures. We do not hide:

- the pool's total holdings;
- each operator's aggregate earnings, revealed per cohort at withdrawal, with the batch's nullifiers and the epoch roots it spans (a coarse timing fingerprint);
- coarse timing in general;
- the per-cohort totals revealed when an expired bucket is reclaimed;
- that the two outputs of one assignment or redemption are siblings in the same transaction;
- the submitter address (a_{sub}) attached to each assignment and redemption;
- the epoch E of each spend's input note, so a spend's anonymity set is at most that epoch's leaves; because epochs are time-bounded, and a purchase leaf publishes h_{exp} while a spent leaf's expiration is bounded by the note lifetime, E also brackets the input note's expiration cohort;
- the query content an indexer sees, never linked to a querier.

We also do not defend against an adversary who already knows, by external means, that a person controls a particular credit and watches that person's own device or network; such targeted attacks are out of scope here, as for the underlying SimpleX transport.

6 Community Credits

Community Credits let end users fund group channels anonymously, paying the operators that serve the group. They behave like prepaid credit: users buy them via a dApp for stablecoin and hold them as *private credit notes* in the app [19]; only registered operators and the network treasury withdraw the backing stablecoin.

6.1 Background and Related Work

6.1.1 Note-Based Privacy Systems

Community Credits use the note-and-nullifier pattern introduced by Zerocash [14] and deployed in Zcash: commitments represent notes; spending reveals a pseudorandom nullifier and proves membership/authorization in zero knowledge without revealing which note was consumed [15].

Privacy Pools generalize note systems with explicit compliance boundaries; Community Credits take a narrower closed-loop form: withdrawals (stablecoin egress) are operator/network-only [16].

The idea of unlinkable prepaid value redeemed later traces to Chaumian e-cash [29]; locking such a token to a single redeemer, as in pubkey-locked ecash [30], is the closest analogue to the private assignment used here, which additionally hides the purchase the credit came from.

To bound on-chain state, we organize commitments into per-epoch trees and retain their frozen roots, in the spirit of Aztec/Miden-style epoch designs [17, 18].

6.1.2 zkSNARKs (Groth16)

Community Credits require users and communities to generate zero-knowledge proofs on mobile devices, which is decisive for the choice of proof system. zkSTARKs [21] are transparent and post-quantum but need 4–8 GB of RAM and seconds of proving even on flagship phones, and crash on mid-range devices, so they are impractical here. We target Groth16 [22]: under 1.5 GB and 0.1–3 s on mobile, and constant-size proofs with cheap constant-cost on-chain verification, well within an EVM-compatible chain's budget. Its per-circuit trusted setup is a one-time, publicly verifiable MPC, an acceptable cost given the mobile constraint. This governs the client circuits (assignment and redemption on mobile, creation in the purchaser's web dApp); the operator withdrawal proves server-side and uses a folding scheme instead (Section 6.4.1).

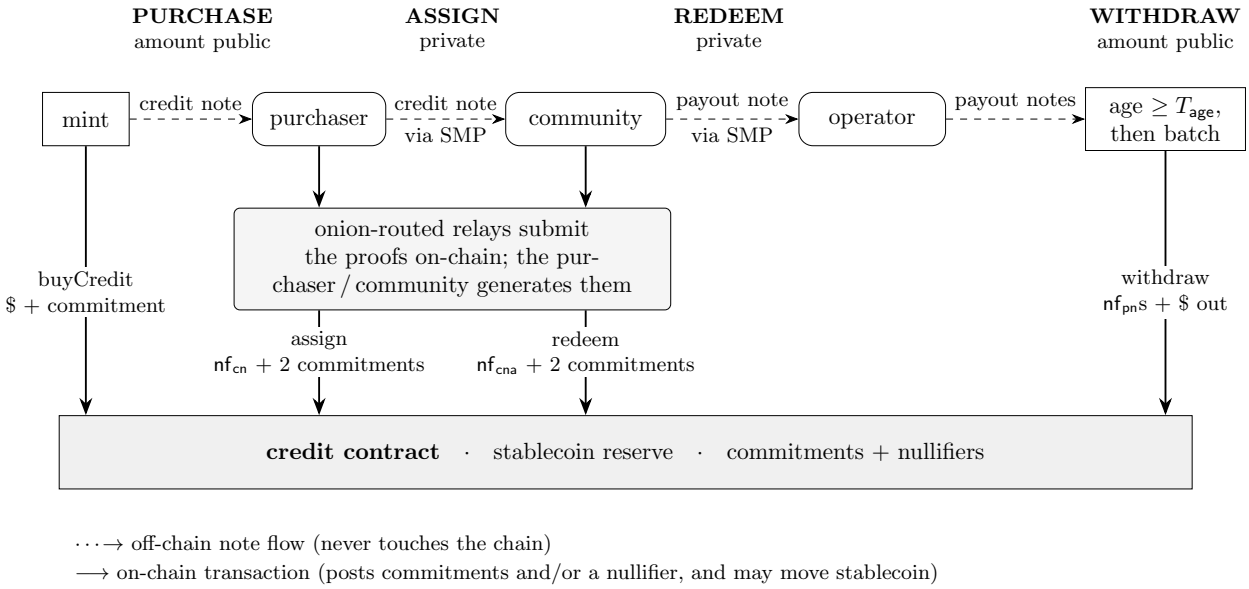

Figure 1: Credit lifecycle, left to right. Notes are handed to counterparties out-of-band (dashed); each on-chain transaction posts only commitments and/or a nullifier, and may move stablecoin (solid). Assignment and redemption are relayed by submitters, which submit but do not generate the proofs; withdrawal waits until each payout note reaches age T_{age} and is batched.

6.2 Design Overview

Figure 1 traces a credit from purchase to withdrawal across the three roles. Unlike privacy pools that publish an encrypted note ciphertext on-chain for wallets to scan, Community Credits deliver notes *out-of-band over SimpleX SMP* (Section 6.3.3); the contract stores only commitments, nullifiers, and aggregate bucket state, never encrypted notes.

6.2.1 Roles

Users (purchasers) Hold private credit notes delivered out-of-band (for example by QR code) and assign them to communities with zkSNARK proofs; their community membership stays unlinkable to everyone.

Communities (redeemers) Receive assigned credits and redeem them with operators via zkSNARK proofs; a community is unattributable on-chain, though it may be known to its own operator out-of-band.

Operators Run SimpleX messaging servers and, with the treasury, are the only parties that may withdraw stablecoin, and only against previously accrued on-chain claims. A public, registered set, not anonymous; only their aggregate turnover is public, while which communities they serve stays hidden.

Network treasury (governance) Receives a fixed share of each withdrawal and the unredeemed remainder of each expired cohort, funding development, network-common costs, and the submission cashback pot (Section 6.3.6); everything it receives is an aggregate. It is controlled by the SimpleX Network Consortium.

Credit contract Receives stablecoin, issues (mints) Community Credit notes, holds stablecoin reserves, maintains commitment trees and nullifier sets, verifies *creation, assignment, redemption, and withdrawal* proofs, enforces permissioning, and applies revenue sharing. A purchase atomically deposits its stablecoin backing and mints the note in one transaction. These functions can be split between several contracts.

Indexers Run a blockchain full node, follow credit events, and serve chain state and Merkle authentication paths to clients over onion-routed SimpleX requests (Section 6.5). This is a role, not a separate party: an operator that runs a full node may also serve as an indexer, and where operators do not, indexers are a dedicated infrastructure role.

6.3 Protocol

Table 1 collects the notation used throughout this section.

Let h be the current block height. We group block heights into *buckets* of a fixed *cadence* Δ_{bucket} blocks (a quarter): *expiration buckets* group credits by expiry, and *submission buckets* group nullifiers by spend time (Section 6.3.7). The client aligns each credit's expiration to a bucket boundary, so a wide range of purchases shares one published height. Let $b(h)$ be the bucket index containing h. A note with expiration height h_{exp} belongs to expiration bucket $e = b(h_{\mathsf{exp}})$; the notes sharing a bucket form a *cohort*, the unit of liability, withdrawal, and reclaim. Distinct from a bucket, an *epoch* (Section 6.3.9) is the append-only commitment tree that holds notes until it freezes, and it sets a spend's anonymity set. See Figure 2.

6.3.1 Keys and Credit Format

Each user generates a spending key sk, a uniformly random field element. The corresponding "public key" is the Poseidon-based hash $\mathsf{pk} = \mathsf{Poseidon}(\mathsf{sk})$. We deliberately avoid an elliptic-curve-based keypair (Baby Jubjub [23] $\mathsf{pk} = \mathsf{sk} \cdot G$): an in-circuit scalar multiplication would dominate the assign and redeem circuits, whereas a Poseidon-based ownership hash costs a single hash evaluation, keeping those circuits within the mobile proving budget of Section 6.1.2. The security argument relies on four standard properties of Poseidon together with Groth16 soundness, each supporting a distinct guarantee:

- preimage-resistance (one-wayness) keeps sk hidden behind $\mathsf{pk} = \mathsf{Poseidon}(\mathsf{sk})$ and the published nullifier;
- keyed pseudorandomness of $\mathsf{Poseidon}(\mathsf{sk}, \cdot)$ makes nullifiers unlinkable across a holder's notes;
- hiding of the Poseidon commitment under the uniform ρ conceals note contents;
- collision-resistance binds each commitment, which together with sk being pinned by pk inside the commitment gives nullifier uniqueness, so no note can be double-spent.

We never need EC-based digital signatures because note authorization is purely an in-circuit knowledge-of-preimage statement.

A credit note is:

$$\mathsf{note} = (v, h_{\mathsf{exp}}, \mathsf{pk}, \rho, \mathsf{assigned}),$$

where v is value, h_{exp} is the expiration height, ρ is uniformly random commitment randomness, pk is the owner key, and $\mathsf{assigned} \in \{0, 1\}$ indicates whether the credit has been assigned to a community. An unassigned note is owned by its purchaser (owner key $\mathsf{pk} = \mathsf{pk_p}$); an assigned note is earmarked to a community by setting its owner key to that community's published key $\mathsf{pk} = \mathsf{pk_r}$, so only the community, which holds the matching $\mathsf{sk_r}$, can redeem it, and the purchaser who minted it cannot. The community publishes $\mathsf{pk_r}$ as its stable donation handle that many purchasers can

Table 1: Notation.

Symbol	Meaning
Block heights, buckets, and windows	
h, h_{now}	current block height
h_{exp}	note expiration height
h_{create}, h_{assign} h_{redeem}	note creation, assign, redeem height
$b(h)$	bucket index containing height h
$e = b(h_{\mathsf{exp}})$	expiration bucket, i.e. cohort, of a note
$B = b(h_{\mathsf{spend}})$	submission bucket (at assign/redeem)
$e_{\mathsf{now}} = b(h_{\mathsf{now}})$	current bucket index
T_{life}	note lifetime
T_{age}	withdrawal age floor (blocks)
W_{final}	post-expiry finalization window (buckets)
W_{nullset}	nullifier-retention window (buckets)
Δ_{bucket}	bucket cadence (blocks). Same for expiry and submission buckets
δ	freshness grace, $\delta < \Delta_{\mathsf{bucket}}$
Δ_{span}	max blocks an epoch stays unfrozen
Keys and identities	
sk, $\mathsf{pk} = \mathsf{Poseidon}(\mathsf{sk})$	a note's owner key ($\mathsf{pk_p}$ if unassigned, $\mathsf{pk_r}$ if assigned)
$\mathsf{sk_p}$, $\mathsf{pk_p}$	purchaser keys
$\mathsf{sk_r}$, $\mathsf{pk_r}$	community (redeemer) keys; $\mathsf{pk_r}$ the published donation handle
$\mathsf{sk_o}^{(e)}$, $\mathsf{pk_o}^{(e)} = \mathsf{Poseidon}(\mathsf{sk_o}^{(e)})$	operator per-cohort key, identifier
a_{sub}	bound cashback beneficiary (submitter)
Note contents	
v, v_{dest}, v_{change}, v_{out}	note values
ρ, s	note / payout-note randomness
Commitments, nullifiers, trees	
$\mathsf{cm_{cn}}/\mathsf{cm_{cna}}$, $\mathsf{cm_{pn}}$	unassigned/assigned credit-note, payout-note commitment
$\mathsf{cm_{dest}}$, $\mathsf{cm_{change}}$, $\mathsf{cm'}$	assignment / redemption output commitments
$\mathsf{nf_{cn}}/\mathsf{nf_{cna}}$, $\mathsf{nf_{pn}}$	unassigned/assigned credit-note, payout-note nullifier
H_{nf}	digest binding a withdrawal's payout nullifiers
E, R_E, T_E	epoch (append-only tree), its root, its tree
C	epoch leaf capacity
On-chain accounting	
$\mathsf{minted}[e]$, $\mathsf{redeemed}[e]$	liability-bucket counters
a, n	withdrawal subtotal, batch size
a_o, a_T	operator, treasury shares of a withdrawal
v_{reclaim}	unredeemed liability reclaimed at expiry
Parameters and contracts	
M	minimum spend / change (spam floor)
$\mathcal{D}$	fixed denomination set
c, f	submission cashback, pot-funding fraction
V, S	credit contract, backing stablecoin

fund, and distinct randomness ρ keeps its notes mutually unlinkable on-chain. Credits are thus non-transferable: only the assigned community redeems, and the $\mathsf{assigned} = 1$ flag prevents an assigned note from being re-assigned. The on-chain commitment is

$$\mathsf{cm_{cn}} = \mathsf{Com}(\mathsf{note})\ (\mathsf{assigned} = 0), \qquad \mathsf{cm_{cna}} = \mathsf{Com}(\mathsf{note})\ (\mathsf{assigned} = 1),$$

where Com is a Poseidon commitment over the note fields, hiding under the uniform ρ and binding by collision-resistance; we tag the commitment by the assigned flag, $\mathsf{cm_{cn}}$ for an unassigned note and $\mathsf{cm_{cna}}$ for an assigned one. Credit-note and payout-note commitments use domain-separated Poseidon instances, a constant per-family tag prepended as an extra input, so a credit-note leaf can never be reinterpreted as a payout note even though both hold five field elements; the two nullifier families are separated likewise. Every proof statement also commits to the chain id and contract address, binding it to this deployment (Tongo-style [20]); we omit this domain separation from the specifications below. The nullifier is the Poseidon of the note's own commitment,

$$\mathsf{nf_{cn}} = \mathsf{Poseidon}(\mathsf{sk}, \mathsf{cm_{cn}}), \qquad \mathsf{nf_{cna}} = \mathsf{Poseidon}(\mathsf{sk}, \mathsf{cm_{cna}}),$$

revealed when the credit is spent: $\mathsf{nf_{cn}}$ at assignment (spending the unassigned note) and $\mathsf{nf_{cna}}$ at redemption (spending the assigned note).[2]

Each operator likewise holds, for each expiration cohort e, an independently generated withdrawal key $\mathsf{sk_o}^{(e)}$, a uniformly random field element, with public identifier $\mathsf{pk_o}^{(e)} = \mathsf{Poseidon}(\mathsf{sk_o}^{(e)})$, registered per cohort in the operator registry. An active operator self-registers each cohort identifier $\mathsf{pk_o}^{(e)}$, so the consortium cannot silently withhold an operator's future cohorts. Payout notes for cohort e (Section 6.3.5) name $\mathsf{pk_o}^{(e)}$, the operator proves knowledge of $\mathsf{sk_o}^{(e)}$ to withdraw them, and the payout-note salt s is a uniformly random field element. The per-cohort keys are independent, not derived from one master key, so an operator can delete $\mathsf{sk_o}^{(e)}$ once cohort e is settled, bounding the reach of a later key compromise (Section 6.3.6). The trade-off is that a key lost before settlement forfeits that cohort's un-withdrawn value.

All value fields (v, v_{dest}, v_{change}, v_{out}) are range-constrained in-circuit to a fixed bit width whose maximum value is well below the field modulus at creation, assignment, and redemption, so value conservation cannot be met by wrapping modulo the field; the solvency argument depends on this.

Out-of-band note payloads (SimpleX delivery). A purchaser mints its own note, but the recipient of a handoff does not: at assignment the community receives the destination note, and at redemption the operator receives the payout note, each delivered out-of-band over SimpleX.

Hidden redemption value via payout notes. A redemption does not publish its value or name the operator it pays. Instead it emits a sealed *payout note* (a Poseidon commitment to the redeemed value, the beneficiary operator, a salt, the expiration bucket, and the redemption block height) inserted into the commitment tree like any other note. The paying operator receives the opening out-of-band over SimpleX, so it learns the value immediately, while the chain stores only the commitment. The operator's redeemed value is thus held as hidden notes and settled only in aggregate at withdrawal (Section 6.3.6); no per-redemption amount or operator ever appears on-chain. An operator's funds depend on the secrecy of its withdrawal keys (defined above), which it uses in every withdrawal proof; a compromised key, together with the operator's stored note

[2] Compared to a separate nullifier key $\mathsf{nk} = \mathsf{Poseidon}(\mathsf{sk})$ followed by $\mathsf{nf_{cn}} = \mathsf{PRF}_{\mathsf{nk}}(\mathsf{cm_{cn}})$, this single-hash form is equivalent for our purposes under the same PRF assumption on $\mathsf{Poseidon}(\mathsf{sk}, \cdot)$, and saves an in-circuit hash; we do not use nullifier-key separation.

openings and membership paths, lets a thief withdraw that cohort's unspent payout notes, and the key alone recomputes their payout nullifiers, tying those notes to the operator, though linking them to the specific communities served requires the operator's out-of-band records. The per-cohort, independent keys bound this, though only partially (Section 6.3.6).

6.3.2 On-chain State

The credit contract (or set of modular contracts) maintains:

- **Stablecoin reserve.** Holdings of a fungible token S (e.g., USDT or USDC) with an ERC20 interface.
- **Stablecoin accounting.** Two monotone counters:

  $$\mathsf{deposited}, \quad \mathsf{withdrawn},$$

  satisfying:

  $$\mathsf{balance}_S(V) = \mathsf{deposited} - \mathsf{withdrawn}.$$

- **Operator registry.** Authorized operators with payout addresses, per-cohort withdrawal identifiers $\mathsf{pk}_{\mathsf{o}}^{(e)}$, and status (active/frozen), plus revenue-share parameters.
- **Epoch-based commitment trees.** Append-only commitment trees per epoch, implemented as incremental Merkle trees with on-chain frontier, and a mapping $E \mapsto R_E$ from each frozen epoch to its root (Section 6.3.9).
- **Submission-bucketed nullifier sets.** Mapping from a *submission* bucket B (the bucket containing the height at which a note was spent) to sets $\mathsf{nullset}[B]$ of used nullifiers, pruned once no unexpired note can reproduce them (Section 6.3.7). Keying nullifiers by spend time rather than by the spent note's expiration is what lets the expiration stay a private witness at assignment and redemption.
- **Liability buckets.** For each expiration bucket e:

  $$\mathsf{minted}[e], \quad \mathsf{redeemed}[e],$$

  both plaintext: $\mathsf{minted}[e]$ is the outstanding face value whose notes expire in e, and $\mathsf{redeemed}[e]$ accumulates the per-bucket subtotals proved at operator withdrawal (Section 6.3.6). No per-redemption amount is recorded, only the blended withdrawal subtotals.
- **Payout-note nullifiers.** A per-cohort set of payout-note nullifiers consumed at withdrawal, so each payout note is withdrawn at most once; because cohort-e withdrawals are rejected once the finalization window closes, the set for e is deleted at `reclaimExpired`(e) (Section 6.3.8), keeping this state bounded like the submission nullsets. Operators hold their redeemed value as hidden payout notes in the commitment tree, not as a plaintext balance; revenue sharing is applied when those notes are withdrawn (Section 6.3.6).
- **Submission cashback.** A count, per submitter address, of the valid assignment and redemption submissions it has included (Sections 6.3.4, 6.3.5), against which the submitter later claims a gas refund from the treasury-funded cashback pot (Section 6.3.6).

6.3.3 Purchase

A credit is bought in one transaction that deposits its backing and mints the note together, so minted face value is always fully backed and there is no separate funding step. The purchaser uses a web dApp connected to their blockchain wallet. Client-side, the dApp generates the purchaser keys $\mathsf{sk_p}$, $\mathsf{pk_p} = \mathsf{Poseidon}(\mathsf{sk_p})$ and the note $\mathsf{note} = (v, h_{\mathsf{exp}}, \mathsf{pk_p}, \rho, 0)$, with $h_{\mathsf{exp}} = h_{\mathsf{now}} + T_{\mathsf{life}}$ raised to its bucket boundary and commitment $\mathsf{cm_{cn}} = \mathsf{Com}(\mathsf{note})$. It then produces a creation proof π_{create} that $\mathsf{cm_{cn}}$ commits to the public (v, h_{exp}). The wallet then submits

$$\texttt{buyCredit}(\mathsf{cm_{cn}}, v, h_{\mathsf{exp}}, \pi_{\mathsf{create}}),$$

transferring v units of S from the purchaser into the contract. Once it lands, the dApp hands the note and $\mathsf{sk_p}$ to the purchaser's SimpleX Chat app out-of-band, for example by QR code, and the app thereafter assigns and redeems the credit. Purchasers without on-chain stablecoin may instead pay an intermediary off-chain (for example, by card); the intermediary is given $(\mathsf{cm_{cn}}, v, h_{\mathsf{exp}}, \pi_{\mathsf{create}})$ but never sk, so it fronts the stablecoin and submits the call yet cannot spend the credit or tie it to any later assignment (the nullifier $\mathsf{Poseidon}(\mathsf{sk}, \mathsf{cm_{cn}})$ is uncomputable to it). The card path is an optional extension; the trust placed in its intermediary, and the resulting threat model, are out of scope for this paper.

Specification 1: Creation (mint) proof

Public inputs. $(\mathsf{cm_{cn}}, v, h_{\mathsf{exp}})$.

Private inputs (witnesses). The note's owner key $\mathsf{pk_p}$ and randomness ρ.

Statement (proved in zkSNARK). The prover (the purchaser's web dApp) shows:

- note well-formedness: $\mathsf{note} = (v, h_{\mathsf{exp}}, \mathsf{pk_p}, \rho, 0)$ and $\mathsf{cm_{cn}} = \mathsf{Com}(\mathsf{note})$ (unassigned).

Contract checks.

1. the caller transfers v units of S into the contract (the backing);
2. h_{exp} is consistent with T_{life} and bucket-aligned, with any tolerance no larger than the bucket-alignment slack already counted in W_{nullset} (Section 6.3.7);
3. fixed denomination: $v \in \mathcal{D}$;
4. the current epoch has space, rolling over to a new epoch if it is full or past its span (Section 6.3.9);
5. verify the proof π_{create}.

State updates (on success).

1. $\mathsf{deposited} \leftarrow \mathsf{deposited} + v$;
2. append $\mathsf{cm_{cn}}$ to the current epoch tree;
3. with $e = b(h_{\mathsf{exp}})$: $\mathsf{minted}[e] \leftarrow \mathsf{minted}[e] + v$, and initialize $\mathsf{redeemed}[e] = 0$ if e is new;
4. emit $\texttt{CreditCreated}(\mathsf{cm_{cn}}, v, h_{\mathsf{exp}})$ as an on-chain receipt.

6.3.4 Assignment

Assignment transfers a credit (or a portion of it) from a purchaser to a community. It consumes one unassigned input note and produces two output notes: one assigned to the community (destination) and one unassigned (change) returned to the purchaser. Both outputs preserve the original expiration height.

Let the input note be $\mathsf{note}_{\mathsf{in}} = (v, h_{\mathsf{exp}}, \mathsf{pk}, \rho, 0)$ with commitment $\mathsf{cm}_{\mathsf{cn}}$ in epoch tree T_E under root R_E. Let the two output notes be:

$$\mathsf{note}_{\mathsf{dest}} = (v_{\mathsf{dest}}, h_{\mathsf{exp}}, \mathsf{pk}_{\mathsf{r}}, \rho_{\mathsf{dest}}, 1),$$
$$\mathsf{note}_{\mathsf{change}} = (v_{\mathsf{change}}, h_{\mathsf{exp}}, \mathsf{pk}, \rho_{\mathsf{change}}, 0),$$

where pk_{r} is the community's published owner key, with commitments $\mathsf{cm}_{\mathsf{dest}}$ and $\mathsf{cm}_{\mathsf{change}}$, satisfying $v_{\mathsf{dest}} + v_{\mathsf{change}} = v$. Full assignment uses $v_{\mathsf{change}} = 0$.

Specification 2: Assignment proof

Public inputs. $(E, R_E, \mathsf{nf}_{\mathsf{cn}}, h_{\mathsf{now}}, \mathsf{cm}_{\mathsf{dest}}, \mathsf{cm}_{\mathsf{change}}, a_{\mathsf{sub}})$, where a_{sub} is the address entitled to the submission cashback.

Private inputs (witnesses). $(\mathsf{sk}, \mathsf{note}_{\mathsf{in}}, \mathsf{path}, \rho_{\mathsf{dest}}, \rho_{\mathsf{change}}, \mathsf{pk}_{\mathsf{r}}, v_{\mathsf{dest}})$ and the expiration height h_{exp}.

Statement (proved in zkSNARK). The prover (purchaser) shows:

- $\mathsf{cm}_{\mathsf{cn}} = \mathsf{Com}(\mathsf{note}_{\mathsf{in}})$ and $\mathsf{cm}_{\mathsf{cn}} \in T_E$ with root R_E;
- ownership $\mathsf{pk} = \mathsf{Poseidon}(\mathsf{sk})$ and nullifier $\mathsf{nf}_{\mathsf{cn}} = \mathsf{Poseidon}(\mathsf{sk}, \mathsf{cm}_{\mathsf{cn}})$;
- input is unassigned: $\mathsf{assigned} = 0$;
- not expired: $h_{\mathsf{exp}} \geq h_{\mathsf{now}}$ (so h_{exp} stays a private witness);
- conservation: $v_{\mathsf{dest}} + v_{\mathsf{change}} = v$, with $v_{\mathsf{dest}} \geq M$ and $v_{\mathsf{change}} \in \{0\} \cup [M, \infty)$, for a minimum spend M (a spam floor);
- output correctness: $\mathsf{cm}_{\mathsf{dest}} = \mathsf{Com}(\mathsf{note}_{\mathsf{dest}})$ with $\mathsf{assigned} = 1$ and owner key pk_{r} (the community's published key, embedded but not otherwise constrained, so a wrong value only self-harms the purchaser);
- change correctness: $\mathsf{cm}_{\mathsf{change}} = \mathsf{Com}(\mathsf{note}_{\mathsf{change}})$ with $\mathsf{assigned} = 0$;
- expiration preservation: both outputs keep the input's h_{exp}.

Contract checks.

1. verify R_E is the stored root for epoch E, or lies in the retained recent-roots window for a live epoch (Section 6.3.9);
2. check h_{now} is recent: $h_{\mathsf{now}} \leq$ the inclusion block height and within δ of it (never in the future);
3. with submission bucket B the bucket of the inclusion block height, check $\mathsf{nf}_{\mathsf{cn}}$ is unused in every active submission bucket (Section 6.3.7);
4. verify the proof π_{assign};
5. insert $\mathsf{nf}_{\mathsf{cn}}$ into $\mathsf{nullset}[B]$;
6. ensure the current epoch has space for both output leaves (a pair never straddles two epochs), rolling over to a new epoch if it is full or past its span (Section 6.3.9);
7. append $\mathsf{cm}_{\mathsf{dest}}$ and $\mathsf{cm}_{\mathsf{change}}$ to the current epoch tree;
8. credit cashback: require $\mathsf{msg.sender} = a_{\mathsf{sub}}$ and increment a_{sub}'s claim by one assignment (Section 6.3.6); binding the beneficiary into the proof stops a mempool observer from replaying the payload to redirect the cashback.

Assignment moves no stablecoin and only restructures ownership among the private notes; the bucket liability $\mathsf{minted}[e]$ is unchanged, since the destination and change notes preserve the input's expiration and sum to its value. The purchaser delivers the destination note payload to the community out-of-band over SimpleX. On receipt the community recomputes the commitment

from the payload using its *own* $\mathsf{pk_r}$ (a mis-addressed note then fails the check), confirms that leaf is in the tree under a finalized root (a soft-head inclusion is reorg-revertible, so it treats the donation as received only against finalized state), checks the remaining lifetime is acceptable and the payload has $\mathsf{assigned} = 1$ (an unassigned note is not redeemable as delivered), and rejects any note whose commitment it has already accepted: two byte-identical notes share one nullifier, so only one is ever spendable, and without this check a replayed payload would be double-counted as two donations. The prover-named submitter a_{sub} that relays the assignment accrues a claimable cashback (Section 6.3.6); the floor M (each assignment and its change move at least M) bounds how much cashback a credit's value can draw.

6.3.5 Redemption

A redemption consumes one *assigned* input note and produces two outputs: a change note to the same owner (community) and a sealed *payout note* that holds the redeemed value for the operator. Full redemption is $v_{\mathsf{change}} = 0$; partial redemption has $0 < v_{\mathsf{change}} < v$.

Let the input note be $(v, h_{\mathsf{exp}}, \mathsf{pk}, \rho, 1)$, owned under the community's key $\mathsf{pk} = \mathsf{pk_r}$, with commitment $\mathsf{cm_{cna}}$ included in epoch tree T_E under root R_E. Let the change note be $\mathsf{note}' = (v_{\mathsf{change}}, h_{\mathsf{exp}}, \mathsf{pk}, \rho', 1)$ with commitment cm', keeping the same owner. Define:

$$v_{\mathsf{out}} + v_{\mathsf{change}} = v, \qquad v_{\mathsf{out}} \geq M, \quad v_{\mathsf{change}} \in \{0\} \cup [M, \infty).$$

Let $e = b(h_{\mathsf{exp}})$ and let h_{redeem} be the current block height. The payout note commits to the redeemed value, the beneficiary operator, a uniformly random salt s, the bucket, and the redemption height:

$$\mathsf{cm_{pn}} = \mathsf{Poseidon}(v_{\mathsf{out}}, \mathsf{pk_o}^{(e)}, s, e, h_{\mathsf{redeem}}),$$

which the community sends, opened, to the operator out-of-band over SimpleX. The operator applies the same acceptance rule as a community receiving a note (Section 6.3.4), including its finalized-root requirement, and renders service only against finalized state; it adds two checks: the payout note must name a $\mathsf{pk_o}^{(e)}$ whose cohort key the operator still holds, and whose finalization window leaves time to withdraw.

No operator is named, no amount is published, no expiration cohort is revealed beyond the epoch bracket, and no plaintext balance is credited; the redeemed value lives only in the payout note. Submitting the redemption immediately is what secures the spend against a re-spent note (the nullifier is claimed on-chain at once), and the submission cashback pays whoever submits it, so the operator need never be the transaction sender. The chain cannot enforce that it is not, however, since $\mathsf{pk_o}^{(e)}$ is private: an operator that relays its own redemptions to collect the cashback links its submitter address to its later withdrawals by timing and volume, weakening its own unlinkability. No stablecoin leaves the pool during redemption; it leaves only at withdrawal, where revenue sharing is enforced (Section 6.3.6).[3]

6.3.6 Withdrawal

Redemption credits no plaintext balance; the operator's value is held in hidden payout notes (Section 6.3.5). To cash out, an operator folds a set of its payout notes for one expiration bucket e into

[3]The spend statement does not verify that $\mathsf{pk_o}^{(e)}$ is a registered operator, since the registry is dynamic on-chain state and $\mathsf{pk_o}^{(e)}$ is private at redemption. Registration is instead enforced at withdrawal: a redemption naming an unregistered $\mathsf{pk_o}^{(e)}$ yields a payout note that no one can withdraw, and its value forfeits to the treasury at reclaim. Binding $\mathsf{pk_o}^{(e)}$ to a committed registry root in-circuit would close this at extra proof cost, and is left as a known limitation.

Specification 3: Redemption (spend) proof

Public inputs. $(E, R_E, \mathsf{nf}_{\mathsf{cna}}, h_{\mathsf{redeem}}, \mathsf{cm}', \mathsf{cm}_{\mathsf{pn}}, a_{\mathsf{sub}})$, with h_{redeem} the redemption height doubling as the freshness reference and a_{sub} the cashback-entitled address.

Private inputs (witnesses). $(\mathsf{sk}, \mathsf{note}, \mathsf{path}, \rho', \mathsf{pk}_{\mathsf{o}}^{(e)}, s)$, the redeemed value v_{out}, and the expiration height h_{exp} (hence its bucket e); beyond the epoch bracket, the cohort e becomes visible only in aggregate at withdrawal.

Statement (proved in zkSNARK). The prover (community) shows:

- $\mathsf{cm}_{\mathsf{cna}} = \mathsf{Com}(v, h_{\mathsf{exp}}, \mathsf{pk}, \rho, 1)$ and $\mathsf{cm}_{\mathsf{cna}} \in T_E$ with root R_E;
- ownership $\mathsf{pk} = \mathsf{Poseidon}(\mathsf{sk})$, so the prover holds the community key the note was earmarked to, and nullifier $\mathsf{nf}_{\mathsf{cna}} = \mathsf{Poseidon}(\mathsf{sk}, \mathsf{cm}_{\mathsf{cna}})$;
- assignment: $\mathsf{assigned} = 1$ (redeemable, not re-assignable);
- change correctness: $\mathsf{cm}' = \mathsf{Com}(v_{\mathsf{change}}, h_{\mathsf{exp}}, \mathsf{pk}, \rho', 1)$;
- payout-note correctness: $\mathsf{cm}_{\mathsf{pn}} = \mathsf{Poseidon}(v_{\mathsf{out}}, \mathsf{pk}_{\mathsf{o}}^{(e)}, s, e, h_{\mathsf{redeem}})$ with $e = b(h_{\mathsf{exp}})$ in-circuit, where $\mathsf{pk}_{\mathsf{o}}^{(e)}$ is the operator's registered cohort-e identifier the community was given out-of-band (its validity not checked here);
- not expired: $h_{\mathsf{exp}} \geq h_{\mathsf{redeem}}$, in-circuit so h_{exp} is not revealed;
- conservation: $v_{\mathsf{out}} + v_{\mathsf{change}} = v$ with $v_{\mathsf{out}} \geq M$ and $v_{\mathsf{change}} \in \{0\} \cup [M, \infty)$.

Contract checks. Submitted by the prover-named submitter a_{sub}; the transaction never names the operator.

1. verify R_E is the stored root for epoch E, or lies in the recent-roots window for a live epoch (Section 6.3.9);
2. check h_{redeem} is recent: $h_{\mathsf{redeem}} \leq$ the inclusion block height and within δ of it (never in the future; the reference for $h_{\mathsf{exp}} \geq h_{\mathsf{redeem}}$);
3. with submission bucket B the bucket of the inclusion block height, check $\mathsf{nf}_{\mathsf{cna}}$ is unused in every active submission bucket (Section 6.3.7);
4. verify the proof (Section 6.4.1);
5. insert $\mathsf{nf}_{\mathsf{cna}}$ into $\mathsf{nullset}[B]$;
6. ensure the current epoch has space for both output leaves (a pair never straddles two epochs), rolling over to a new epoch if it is full or past its span (Section 6.3.9);
7. append cm' and $\mathsf{cm}_{\mathsf{pn}}$ to the current epoch tree;
8. credit cashback: require $\mathsf{msg.sender} = a_{\mathsf{sub}}$ and increment a_{sub}'s claim by one redemption (Section 6.3.6); the bound beneficiary makes the payload safe to relay in a public mempool without the cashback being front-run.

a single withdrawal proof (Section 6.4.1) and submits it under its public, registered identity.

The withdrawal reveals $(\mathsf{pk}_{\mathsf{o}}^{(e)}, e, n, a)$, the batch's n payout nullifiers, and the epoch roots it spans (a coarse timing fingerprint). The individual amounts $v_{\mathsf{out},i}$ stay hidden (the folding scheme is zero-knowledge and its intermediate instances are never published, Section 6.4.1), but the age floor sets no minimum on n, so for a low-volume operator n can be small and its subtotal approaches a single amount. The threat model assumes this low-volume case away. The key $\mathsf{sk}_{\mathsf{o}}^{(e)}$ belongs to cohort e alone, so once the operator settles and deletes it, a later compromise cannot recompute that cohort's payout nullifiers or reach any other cohort's notes; the identifier $\mathsf{pk}_{\mathsf{o}}^{(e)}$ revealed here is likewise cohort-specific. This limit is only partial. A cohort settles only after its notes expire and are withdrawn, so its key must be held for roughly $T_{\mathsf{life}} + W_{\mathsf{final}}\Delta_{\mathsf{bucket}}$, and the still-open cohorts thus span roughly the last year or more of redemptions; moreover, deleting the key protects little on its own, since the operator also holds, out-of-band, each payout-note opening and the amount and community of every redemption it served, which identify those redemptions without any key. The Operator Deed therefore requires timely deletion of those openings and records too, not only the cohort key. Freezing an operator prevents it from self-registering new cohort keys and signals its communities, off-chain, to stop redeeming with it, but it is not an on-chain redemption block: a redemption names $\mathsf{pk}_{\mathsf{o}}^{(e)}$ privately and the contract checks registration only at withdrawal, so a redemption to an already-registered cohort of a frozen operator is not refused, and the operator can still withdraw value accrued before or after the freeze (since h_{redeem} is private, the two are indistinguishable on-chain). A frozen operator therefore keeps serving and withdrawing its already-registered cohorts, up to T_{life} ahead. A community that relies on a single operator for its registry view can also be shown a stale "still-active" view of it. Both are residual risks, mitigated by operator diversity and off-chain monitoring rather than by the contract. The finalization window W_{final} must be sized so a note redeemed near its bucket's expiry can still reach age T_{age} before the window closes; Section 6.3.9 gives the full sizing, which also covers the epoch-freeze latency. Because $\mathsf{redeemed}[e]$ is fed only here, expired-fund reclaim (Section 6.3.8) reads it directly once the window W_{final} closes.

6.3.7 Nullifier Bucketing and Garbage Collection

Figure 2 traces one credit across both bucket systems, from purchase to the reclaim of its cohort's expired funds.

Nullifiers are filed by *submission* bucket $B = b(h_{\mathsf{spend}})$, the bucket containing the height at which the note was spent (assigned or redeemed), not by the spent note's expiration. The contract files under its own inclusion block height, not the prover-supplied freshness reference, so the filed bucket is exact. The contract therefore never needs h_{exp} to file or check a nullifier, which is what lets the expiration stay a private witness at assignment and redemption.

Because the filing bucket depends on when a note is spent rather than on the note, a double-spend check cannot look in one deterministic bucket. Each spend instead checks its nullifier against *all active* submission buckets and files it in the current one. This is a constant number of mapping lookups, at most W_{nullset} (defined below), not a scan; equivalently, a single global spent-nullifier set answers the check in $O(1)$ and the buckets serve only as a delete-index for pruning. The number of active buckets is bounded by the lifetime: the contract garbage-collects $\mathsf{nullset}[B]$ once $b(h_{\mathsf{now}}) \geq B + W_{\mathsf{nullset}}$ with the fixed window $W_{\mathsf{nullset}} = \lceil T_{\mathsf{life}}/\Delta_{\mathsf{bucket}} \rceil + 3$ buckets, sized so that no unexpired note can reproduce a garbage-collected nullifier and both spends of any one note fall inside the window, keeping the double-spend check complete; at most W_{nullset} buckets are ever

Specification 4: Operator withdrawal proof (folded over an arbitrary set, Section 6.4.1)

Public inputs (revealed). $(\mathsf{pk}_{\mathsf{o}}^{(e)}, e, n, a)$ with $a = \sum_i v_{\mathsf{out},i}$; a single digest $H_{\mathsf{nf}} = \mathsf{Poseidon}(\mathsf{nf}_{\mathsf{pn},1}, \ldots, \mathsf{nf}_{\mathsf{pn},n})$ binding the n payout nullifiers (the list itself is passed as calldata, not as n proof inputs); the distinct epoch roots used; the withdrawal height h_{now}.

Private inputs (witnesses). $\mathsf{sk}_{\mathsf{o}}^{(e)}$, and for each note its opening $(v_{\mathsf{out},i}, s_i, h_{\mathsf{redeem},i})$ and membership path.

Statement (folded, one step per note; age floor at finalization). The operator shows, over an arbitrary number of notes:

- each $\mathsf{cm}_{\mathsf{pn},i} = \mathsf{Poseidon}(v_{\mathsf{out},i}, \mathsf{pk}_{\mathsf{o}}^{(e)}, s_i, e, h_{\mathsf{redeem},i})$ lies in its frozen epoch tree T_{E_i}, opened under $\mathsf{sk}_{\mathsf{o}}^{(e)}$ with $\mathsf{pk}_{\mathsf{o}}^{(e)} = \mathsf{Poseidon}(\mathsf{sk}_{\mathsf{o}}^{(e)})$, and shares the bucket e;
- each payout nullifier $\mathsf{nf}_{\mathsf{pn},i} = \mathsf{Poseidon}(\mathsf{sk}_{\mathsf{o}}^{(e)}, \mathsf{cm}_{\mathsf{pn},i})$ is well-formed (unlinkable to its note without $\mathsf{sk}_{\mathsf{o}}^{(e)}$);
- **age floor**, checked once at finalization: $h_{\mathsf{now}} - \max_i h_{\mathsf{redeem},i} \geq T_{\mathsf{age}}$, so every note is at least T_{age} old (a time-based floor with no minimum-count requirement, so a withdrawal is never blocked for want of volume; the anonymity set is correspondingly large only when volume is high, Section 7);
- subtotal $a = \sum_i v_{\mathsf{out},i}$.

Contract checks.

1. the operator's cohort-e identifier $\mathsf{pk}_{\mathsf{o}}^{(e)}$ is registered (a frozen operator remains registered and may still withdraw already-accrued cohorts; freezing blocks only the self-registration of new cohort keys, not this check);
2. h_{now} is recent and not in the future ($h_{\mathsf{now}} \leq$ the inclusion block height, within a small window of it), so the age floor $h_{\mathsf{now}} - \max_i h_{\mathsf{redeem},i}$ is measured against the chain and cannot be inflated by a prover-chosen future value;
3. the bucket is still within its post-expiry finalization window, $e_{\mathsf{now}} < e + W_{\mathsf{final}}$ (Section 6.3.8);
4. verify the folded proof, and each distinct epoch root against the stored root for its epoch (Section 6.3.9);
5. recompute H_{nf} from the submitted nullifier list and check it matches the proof's public input; the n payout nullifiers are pairwise distinct and unused; record them;
6. solvency: require $\mathsf{redeemed}[e] + a \leq \mathsf{minted}[e]$, then $\mathsf{redeemed}[e] \leftarrow \mathsf{redeemed}[e] + a$;
7. revenue share: operator share a_o, treasury share $a_T = a - a_o$ (a fixed fraction funds the cashback pot); transfer a_o in S to the operator's registered payout address (fixed by $\mathsf{pk}_{\mathsf{o}}^{(e)}$ in the registry, so the destination is set by the proof, not by the submitter) and a_T to the treasury;
8. $\mathsf{withdrawn} \leftarrow \mathsf{withdrawn} + a$.

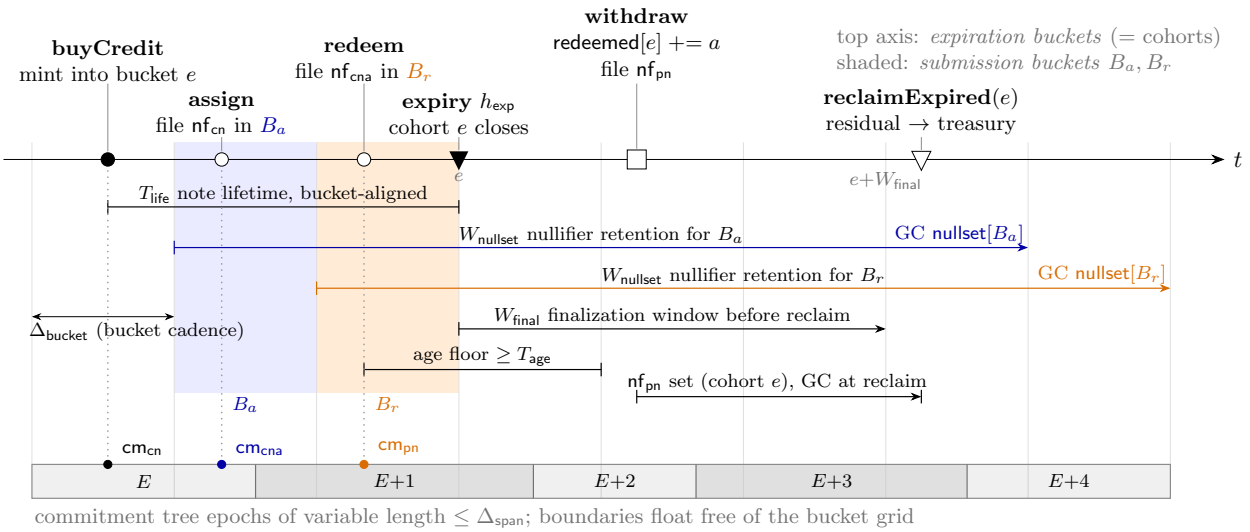

Figure 2: Bucket timeline for one credit; the axis t is block height. Both bucket systems share the cadence Δ_{bucket}: the top axis groups notes by expiry into cohorts, while assignment and redemption file credit-note nullifiers into submission buckets B_a, B_r, each garbage-collected after the retention window W_{nullset}. After expiry, operators withdraw within the finalization window W_{final}, after which `reclaimExpired`(e) *returns cohort e's residual to the treasury. The bottom strip is the commitment-tree* epochs *(Section 6.3.9): variable length ($\leq \Delta_{\mathsf{span}}$), boundaries floating free of the grid. Dotted drops trace each transaction to the commitment it appends, $\mathsf{cm}_{\mathsf{cn}}$ (purchase), $\mathsf{cm}_{\mathsf{cna}}$ (assign), $\mathsf{cm}_{\mathsf{pn}}$ (redeem); each spend also files the matching nullifier nf_τ of the cm_τ note it consumes, with $\mathsf{nf}_{\mathsf{pn}}$ filed at withdrawal into the per-cohort set that reclaim deletes.*

active, so the per-spend check is bounded (Appendix C derives the bound).[4] This deletion is lazy and may be paginated across calls, and running it late is harmless: a bucket past $B + W_{\mathsf{nullset}}$ is no longer among the active buckets a spend consults, so no double-spend check depends on it.

6.3.8 Expiration and Reimbursement

For each bucket e, the contract stores plaintext $\mathsf{minted}[e]$ and $\mathsf{redeemed}[e]$, the latter accumulated from the per-bucket subtotals proved at operator withdrawal (Section 6.3.6).

After bucket e expires, operators have a finalization window of W_{final} buckets to withdraw their e-payout notes; once $e_{\mathsf{now}} \geq e + W_{\mathsf{final}}$ the contract rejects further e-withdrawals, so $\mathsf{redeemed}[e]$ is final. The contract check is given in Specification 5.

Specification 5: Reclaim: `reclaimExpired`(e)

Callable by. Any party, once the finalization window has closed ($e_{\mathsf{now}} \geq e + W_{\mathsf{final}}$).

Contract checks and effects.

1. check $e_{\mathsf{now}} \geq e + W_{\mathsf{final}}$ and that e has not already been reclaimed, recorded in a per-cohort flag that survives the later storage clean-up;
2. compute the unredeemed liability $v_{\mathsf{reclaim}} = \mathsf{minted}[e] - \mathsf{redeemed}[e]$, nonnegative by the withdrawal solvency invariant $\mathsf{redeemed}[e] \leq \mathsf{minted}[e]$ (the value of e-notes never redeemed, plus any e-payout note left un-withdrawn past the window, which forfeits);
3. transfer v_{reclaim} to the treasury (which may route a fraction to the cashback pot) and set $\mathsf{withdrawn} \leftarrow \mathsf{withdrawn} + v_{\mathsf{reclaim}}$.

Steps 1–3 are $O(1)$. Cohort e's payout-nullifier set is then marked for deletion and pruned lazily in separate, paginated calls (and the liability-bucket storage likewise); pruning late is harmless, because the closed finalization window already rejects any e-withdrawal, so the inert set is never consulted again. Submission nullsets are pruned separately (Section 6.3.7).

The reclaimed figure is an aggregate per cohort: $\mathsf{minted}[e]$ (public) minus a sum of blended withdrawal subtotals, never a per-credit amount.

Both nullifier garbage collection (Section 6.3.7) and `reclaimExpired` cost gas with no protocol reward, so a treasury-funded keeper triggers them.

6.3.9 Epoch-Based Commitment Trees

Commitments are organized into per-epoch append-only Merkle trees T_E. Only the current epoch tree is writable. An epoch ends when its fixed-depth tree fills or after a maximum block span Δ_{span}, whichever comes first; the first transaction that would overflow the tree (or the first past the span) freezes the current epoch and opens the next in the same call, so under continuous activity no dedicated keeper is needed and the rollover cost falls on one ordinary submitter. Because every spend names its input's epoch E, a spend's anonymity set is at most that epoch's leaves (for a redemption, effectively the epoch's assignment-destination and redemption-change leaves, since purchase leaves are publicly unassigned and the two outputs of one spend are distinguishable); epoch capacity and Δ_{span} are therefore anonymity parameters as well as cost parameters. Flooding an epoch with cheap leaves to shrink the honest anonymity set is possible but economically costly (Appendix B). During a lull, any party may instead call a permissionless `freezeEpoch` once the

[4] This retention is independent of the liability-bucket finalization window W_{final} (Section 6.3.8), which governs $\mathsf{minted}[e]$ and $\mathsf{redeemed}[e]$, not nullifiers.

chain passes the epoch's open height plus Δ_{span} (or the tree is full), so an operator wanting to withdraw can always force a payout note's epoch to freeze within Δ_{span} of its redemption. The finalization window W_{final} (Section 6.3.8) is sized so that $(W_{\mathsf{final}} - 1)\,\Delta_{\mathsf{bucket}} \geq T_{\mathsf{age}} + \Delta_{\mathsf{span}} + \delta$ plus fold-and-submit time, covering the age floor, the worst-case freeze latency, and folding, so no accrued value is stranded unwithdrawable. When epoch E ends, its final root R_E is frozen and stored in an on-chain mapping $E \mapsto R_E$ of frozen roots. A frozen root is needed only while E can still serve a spend or a withdrawal. A note appended in E expires at most $T_{\mathsf{life}} + \Delta_{\mathsf{bucket}}$ after E's freeze height h_{freeze} (its expiration is a private witness, so the bound is read from the append height, not the note), so the latest cohort E can hold a note for is $e^* = b(h_{\mathsf{freeze}}) + \lceil T_{\mathsf{life}}/\Delta_{\mathsf{bucket}} \rceil + 1$, whose payout notes stay withdrawable for a further W_{final} buckets. The contract therefore prunes R_E once $e_{\mathsf{now}} \geq e^* + W_{\mathsf{final}}$: by then every note in E has expired, so no valid spend references R_E, and every cohort E could serve has closed its finalization window, so no withdrawal is stranded. Old roots are thus prunable and the mapping stays bounded, closing the last previously-unbounded component of state. Creation, assignment, and redemption each trigger this rollover when the current epoch is full, so a submission never reverts for lack of space.[5]

Inclusion proofs via indexers. Indexer nodes follow credit events and maintain the epoch trees; a client requests a Merkle path via onion-routed SimpleX messages, `getPath(epoch, root, cm)`, and verifies it against the current root. Indexers serve these membership paths and may relay assignment and redemption payloads, but never the note payloads themselves. A `getPath` does reveal to the indexer which leaf a client is about to spend; for a freshly purchased credit's assignment input that leaf is the public purchase commitment (a re-assigned change note's input leaf is one hop removed), so an indexer that also watches the chain could link a purchase to the assignment it funds by content and timing. Decoy or batched path requests blunt this, and a wallet that instead maintains its own membership witnesses (updating them as commitments are appended, as above) avoids the query for its own notes entirely.

To spend a note created in epoch E, the zkSNARK proves membership in T_E under R_E, and the contract verifies R_E is the stored root for epoch E. A frozen past-epoch root is stable, so most spends prove against a fixed root; for the current epoch, whose root advances on every append, the contract retains a bounded window of recent roots and accepts a proof against any of them, so an honest proof is not invalidated by a concurrent insertion.[6] Proving against a live-epoch recent root does reveal a leaf-count prefix, since the chosen root pins how many leaves precede it: a spend finalized against the live epoch, most visibly a fresh purchase that is immediately assigned, has a smaller effective anonymity set than the epoch's full leaf count, which it regains once the epoch freezes and later spends prove against the single frozen root R_E. The window is retained across a rollover: a just-frozen epoch's recent roots stay acceptable for the same latency period, so a proof built against a live-epoch root whose epoch freezes in flight is not rejected.[7]

[5]The contract stores the current epoch root and a frontier (an incremental Merkle representation) to support efficient on-chain appends, and wallets update their own membership witness as commitments are appended, without revealing which note they own, mirroring Zcash-style incremental commitment trees [15].

[6]The window is sized for the worst-case prove-and-relay latency rather than kept minimal: it must be large enough that a proof built against a recent root survives mobile proving and relay delay, yet bounded so stale roots do not accumulate. Root validity is enforced by the stored epoch roots; spend freshness is enforced separately by the h_{now} recency check and the in-circuit expiry check, not by this intra-epoch buffer.

[7]The contract executes at the soft chain head; clients and submitters reference recent *finalized* roots and heights, and the contract's "recent" windows are sized to absorb the finality lag. Reorg safety then rests on two properties rather than an on-chain finality check: a nullifier insertion and its commitment appends happen in one transaction, so a reorganization reverts them together; and nullifier uniqueness is global within the retention window, which vastly exceeds any reorg depth (Section 6.3.7), so a spend re-included at a different height still collides on any second

6.4 Smart Contract Design

We outline a deployment on an EVM-compatible blockchain or rollup, whose pairing precompiles carry the Groth16 verification. The horizons T_{life}, T_{age}, Δ_{span}, and the buckets are denominated in block heights, so a deployment assumes an approximately constant block time or recalibrates them per chain; because h_{exp} is baked immutably into each note commitment, a later change in block time rescales the effective lifetime of notes already minted.

6.4.1 Proof Verification and Aggregation

Community Credits use Groth16 zkSNARK proofs for credit creation, assignment, and redemption (Specifications 1–3), and for operator withdrawal, the last aggregated by folding (below) rather than a fixed-arity circuit (Specification 4).

The credit contract and its verifiers are immutable and non-upgradeable: there is no administrative key that can move or seize user funds or alter the settlement logic, so custody rests with the contract alone, and migration (for example to a post-quantum-sound proof system, Section 7) is by opt-in redeployment rather than in-place upgrade. On-chain, each buy, assign, and redeem also inserts its commitment(s) into the incremental Merkle tree at several Poseidon hashes per leaf.[8]

Withdrawal aggregation by folding. A fixed-arity batch circuit would force padding and cap a withdrawal at a fixed size, so the withdrawal proof is instead an incrementally-verifiable computation over a *folding scheme* [25], aggregating an arbitrary number of notes in the manner of [28]. A single *step* circuit processes one payout note: membership of $\mathsf{cm}_{\mathsf{pn},i}$ against its *frozen* epoch root R_{E_i} (stable until pruned, so foldable at any time), the opening under the cohort key $\mathsf{sk}_{\mathsf{o}}^{(e)}$, and consistency with the shared bucket e. Each step folds into a running accumulator holding the subtotal a, the count n, the newest redemption height $\max_i h_{\mathsf{redeem},i}$, the distinct epoch roots used, and a running digest H_{nf} of the payout nullifiers. The operator folds the notes one at a time, so prover memory stays at a single step and nothing is padded. A note can be folded as soon as its epoch freezes (a short wait after redemption, since a note in the still-live epoch has no frozen root to prove against yet). The one check that depends on withdrawal time is left to a *finalization* step: it proves the age floor once against the newest note, $h_{\mathsf{now}} - \max_i h_{\mathsf{redeem},i} \geq T_{\mathsf{age}}$ (so every note clears it), and compresses the accumulator into a single succinct proof that exposes H_{nf} as its only nullifier-bearing output. This finalization is zero-knowledge and the intermediate folding instances are never published, so the individual $v_{\mathsf{out},i}$ and $h_{\mathsf{redeem},i}$ stay hidden. On-chain, proof *verification* is then the constant Groth16 check of Section 6.1.2 regardless of n: the n nullifiers enter the proof only through the single binding digest H_{nf} (Specification 4), not as n public inputs that would grow the verifier's input cost. The contract additionally recomputes H_{nf} from the submitted nullifier list, checks the few distinct epoch roots against their stored roots (as for any spend, Section 6.3.9), and records the n payout nullifiers, so the transaction's bookkeeping is $O(n)$ in nullifiers and O(epochs spanned) in roots even where the proof is $O(1)$.[9] This is the one proof generated on the operator's server rather than a mobile device, which makes the heavier folding prover acceptable; creation, assignment, and redemption stay plain Groth16, and folding adds only one fixed wrapper setup, with no per-size ceremony.

attempt. Double-spend detection is therefore unaffected.

[8] Absent a Poseidon precompile, these inserts can rival the proof verification in gas.

[9] Folding a nullifier accumulator into the proof, exposing only an old and new root, would make the bookkeeping $O(1)$ too, at the cost of serializing concurrent withdrawals; we keep the plain nullifier set.

6.4.2 Operational Flows

Credit Purchase Flow

1. In a web dApp connected to their wallet, the purchaser generates sk, the note, and its creation proof client-side, and the wallet submits `buyCredit` (Section 6.3.3), transferring v stablecoin into the contract, which verifies the proof and appends the commitment to the current epoch tree.

2. The dApp hands the note and $\mathsf{sk_p}$ to the purchaser's SimpleX Chat app (for example, by QR code), which thereafter assigns and redeems the credit.

Purchasers paying by card route step 1 through an intermediary that fronts the stablecoin; it is never given sk.

Relayed submission and cashback. A user-submitted transaction reveals the sender's account address and network-layer metadata, so assignments and redemptions are *relayed*: the prover sends the proof and its public inputs to a submitter over SimpleX, and the submitter makes the on-chain call [32, 33]. There is no signature; the zero-knowledge proof itself authorizes the state transition and binds its outputs, so a relayer can submit but cannot alter the payload. The prover names its chosen submitter as the cashback beneficiary a_{sub}, a public input of the proof (Specifications 2, 3), so even in a public mempool a copied payload cannot redirect the cashback to a front-runner: the contract pays only a_{sub}. The submitter learns nothing private (the proof reveals no amount or operator, and no cohort beyond the epoch bracket), and any party the prover names may relay and earn the cashback. A community should choose its submitter at random rather than route through its own operator, so the public per-submitter cashback counter fingerprints no single community or operator.

Submitters are compensated not by a fee taken from note value but by a *cashback*. For each valid assignment or redemption it includes, the contract credits the submitter's address with a claimable refund, drawn from a pot the treasury funds out of the withdrawal revenue split (Section 6.3.6) and holds in the chain's native gas token. A submitter recovers its gas by claiming accrued cashback in its own transaction; c covers that claim's gas, and a submitter batches many submissions into one claim. Anyone may relay, and the cashback pays for it.

The pot is self-financing over time because submissions are bounded by value: a credit of face value v draws cashback against at most $2v/M$ submissions, and routing a fraction $f \geq 2c/M$ of withdrawal value to the pot covers this, so farming is unprofitable (Appendix A gives the derivation and parameter calibration). Self-financing is not instantaneous, however: the pot is replenished largely at reclaim (Section 6.3.8), which lags a submission by up to $T_{\mathsf{life}} + W_{\mathsf{final}}\Delta_{\mathsf{bucket}}$, so its pot manager must observe activity and front enough working capital to cover in-flight submissions. An under-funded pot can be drained by high-rate submissions, at which point accrued cashback still records but its payout, and hence new relaying, stalls; a user who then self-submits reveals its own address. This is a liveness dependency on the treasury, and through the self-submission fallback a privacy one.

6.5 Private Blockchain Access

Several components rely on smart contract state as a single source of truth (the operator registry and the Community Credits contract), and reading it must not leak who is querying or what they query. A conventional light client would reveal both: it fetches state over the blockchain's peer-to-peer network, exposing the client's IP address and the exact storage slots it reads.

SimpleX instead reads state over its own onion-routed relays while keeping the cryptographic verification. The client sends an encrypted query through a forwarding relay to an *indexer*, a SimpleX node that holds the relevant contract state and returns the result with a state proof. The forwarding relay sees the client's address but not the query; the indexer sees the query but not the client; and the client verifies the proof locally against a recent finalized state root (below). Thus the client trusts only the proof, not the indexer, and no single party learns both the querier and the query. Indexers can be run by SMP relay operators (discoverable via the SORC, Definition 1) or by dedicated providers.

Tracking the chain head. A state proof attests *authenticity* but not *recency*: an indexer can replay a stale-but-once-valid proof (a since-frozen operator shown as active), so every proof must be tied to a recent finalized state root the client trusts independently. The mechanism, which tracks the head over the SMP channel the client already polls, is deferred to Appendix D.

7 Analysis and Open Questions

The objectives and privacy goals above are argued informally by the design; formal definitions and proofs are left to future work. This section records the residual limits and the decisions that remain open.

Post-quantum soundness (known limitation). The note and payout commitments are Poseidon hashes, so the confidentiality of amounts and identities is post-quantum: a future quantum adversary that harvests the permanent ledger cannot recover what the commitments hide. This relies on the hiding and one-wayness of the Poseidon commitment against such an adversary: low-entropy or published fields (a denomination v, an expiration height h_{exp}, a bucket e, an assigned note's owner key pk_{r}, a payout note's operator $\mathsf{pk}_{\mathsf{o}}^{(e)}$, and the redemption height h_{redeem}) are protected only by their commitment randomness, the note randomness ρ or the payout-note salt s. These randomness values, like the identity keys sk and $\mathsf{sk}_{\mathsf{o}}^{(e)}$, are single BN254 field elements (~ 254 bits), so under Grover search their post-quantum hiding and preimage margins are field-width-bounded at ~ 127 bits, just below the 128-bit target. Soundness, however, depends on Groth16 over the BN254 pairing,[10] which such an adversary breaks, allowing forged proofs and thus unauthorized minting or spending; the per-circuit trusted setup is a second soundness root, since retained toxic waste likewise forges proofs and could drain the pool. Privacy therefore survives a quantum break while integrity does not. Because soundness is a going-forward property and the credit contract is immutable (Section 6.4.1), the mitigation is to deploy a post-quantum-sound successor contract before cryptographically-relevant quantum computers arrive and let holders migrate their notes to it opt-in; this protects future state, needs no administrative upgrade key over the funds, and does not re-commit the past. Any on-chain encrypted note backup (below) must itself use post-quantum encryption, or it reopens the harvest channel.

Community-key compromise (known limitation). A community's donation handle $\mathsf{pk}_{\mathsf{r}} = \mathsf{Poseidon}(\mathsf{sk}_{\mathsf{r}})$ is long-lived and published, and sk_{r} does not rotate automatically. An attacker

[10] Classical soundness of BN254 is estimated at ~ 100 bits under current tower-number-field-sieve results, below the nominal 128-bit target. We accept this for broad EVM compatibility: BN254 pairing precompiles (EIP-196/197) are available on essentially every EVM chain, whereas the BLS12-381 precompiles (EIP-2537) reached Ethereum mainnet only with the Pectra upgrade of May 2025 and are not yet universal across the L2s and alt-EVM chains this design targets. Migrating to a higher-security curve is an option where that precompile is available.

who obtains $\mathsf{sk_r}$ can recompute the nullifier $\mathsf{nf_{cna}} = \mathsf{Poseidon}(\mathsf{sk_r}, \mathsf{cm_{cna}})$ of every note assigned to the community and match it against the on-chain leaves and spent-nullifier history, retroactively deanonymizing the community's entire assignment and redemption record. This is worse than the operator case: an operator holds independent per-cohort keys $\mathsf{sk_o}^{(e)}$ that it deletes once each cohort settles, so a compromise reaches only the still-open cohorts (Section 6.3.6), whereas a community's single standing key exposes all of its history at once, with no such bound. A mitigation, left to future work, is to rotate the donation handle, publishing a fresh $\mathsf{pk_r}$ per epoch or funding period, so a leaked key exposes only that period's redemptions; the cost is usability, since purchasers must fetch the current handle, and splitting a community across handles fragments its anonymity set and requires each handle to be re-published and re-discovered.

Open decisions.

- **Proof system.** The client circuits (creation, assignment, redemption) assume Groth16, which proves quickly on mobile and verifies cheaply but needs a trusted setup per circuit. A universal-setup system (PLONK / UltraHonk) avoids per-circuit ceremonies at some cost in prover time; the choice turns on mobile prover performance. The operator withdrawal is fixed separately to a folding scheme (Section 6.4.1), since it proves server-side over an arbitrary-length list.

- **Expiration-bucket size.** The expiration height is hidden at assignment and redemption, so a bucket does not narrow an individual spend; its size sets only the granularity of the per-cohort totals disclosed at withdrawal and reclaim, and the span of the submission-nullifier retention window W_{nullset} (Section 6.3.7). A coarser bucket combines more value per cohort but lengthens the nullifier-retention window W_{nullset} (Section 6.3.7) and the delay before expired value is reclaimed.

- **Denomination set.** Credits are issued in a fixed set of denominations at purchase only; assigned and redeemed amounts stay private, floored at M. A unique purchase amount could act as a linking fingerprint: the community admin who receives the assignment, or a relayer that later handles a redemption, could re-link the value to the purchaser off-chain. This is the unique-value linkage Kappos et al. and Quesnelle exploit against Zcash's shielded pool [24, 27]. Fixed denominations remove the fingerprint instead of trusting clients to avoid it. The specific set, balancing usability against anonymity-set size, remains to be chosen.

- **Floor and cashback calibration.** The floor M, the per-submission refund c, and the pot-funding split f (whose self-financing bound $f \geq 2c/M$ is derived in Appendix A) track gas price and the native/stablecoin exchange rate; their joint calibration, and how coarse M may be before it harms usability, remains open.

- **Note recovery.** Whether a holder that loses its device can recover its credits from a single backed-up seed plus a durable encrypted backup of each note, and where that backup lives: on-chain for community notes, or in the user's encrypted SimpleX backup.

- **Operator-registration in the spend circuit.** Redemption does not prove $\mathsf{pk_o}^{(e)}$ is registered (the registry is dynamic state); a mis-addressed payout simply forfeits at reclaim. Proving $\mathsf{pk_o}^{(e)}$-membership against a committed registry root would enforce it at redemption, at extra proof cost.

- **Cashback-pot exhaustion as a privacy dependency.** Relaying is funded from the treasury cashback pot; the bound $f \geq 2c/M$ makes it self-financing over time but not instantaneously (Section 6.3.6), and it is self-financing only at the whole-treasury level, since expired-unredeemed

and unregistered-operator value draws cashback while feeding the pot only through reclaim, not the withdrawal split. If the pot drains faster than it refills, relaying stalls and a user who then self-submits reveals its own address, so exhaustion is a privacy dependency, not only a liveness one.

8 Conclusion

This document sets out Community Credits as the economic system of the SimpleX network: a closed-loop, privacy-preserving way to pay for the infrastructure a sustainable and censorship-resistant network depends on, without the user identification a general-purpose payment system would require. Credits are prepaid, non-transferable, expiring access to network capacity: bought openly, assigned to communities in private, redeemed by those communities with registered operators, and settled under revenue sharing the contract enforces, with every credit backed by stablecoin held in a non-custodial contract and read from the chain through onion-routed state queries with local proof verification. The closed-loop shape is deliberate: it lets the system pay operators while keeping users private, and places it within established regulatory exemptions.

The design is argued to meet its economic and solvency objectives, assignment hides the amount and the recipient community, and redemptions, though visible as events, are attribute-hiding and unlinkable: they name no operator, publish no amount, and reveal no expiration cohort beyond a coarse epoch bracket, with the value settled later at withdrawal unlinkable to any specific redemption. The remaining decisions (the proof system, the expiration-bucket size, the denomination set, the floor and cashback calibration, note recovery, and whether redemption proves operator registration) are recorded as open.

Acknowledgements. The authors thank the StarkWare Research Team for their contributions to the initial design. The authors also thank the Web3 Foundation and the Parity team for their help developing the proof of concept for Community Credits on the Polkadot blockchain.

A Cashback-Pot Management

Submitters are paid a cashback rather than a fee taken from note value (Section 6.3.6), so the pot's solvency rests on submissions being bounded by the value that funds them. With no re-assignment, and redeemed value passing into operator-only payout notes that cannot re-enter the assignment cycle, a credit of face value v admits at most v/M assignments and v/M redemptions, so at most $2v/M$ submissions ever draw cashback against it. Writing c for the refund per submission and f for the fraction of withdrawal value routed to the pot, choosing $f \geq 2c/M$ makes each unit of value contribute at least the cashback it can generate, so the pot stays solvent and farming is unprofitable: a submitter pays back through the split at least what it earns. The refund c and split f are governance parameters the treasury tunes against gas price and the native/stablecoin exchange rate, periodically swapping stablecoin into native to refill the pot, and a coarser floor M lowers the split a given c requires. The bound holds only at the whole-treasury level and over time: expired-unredeemed and unregistered-operator value draws cashback while feeding the pot only through reclaim, not the withdrawal split, and reclaim lags a submission by up to $T_{\mathsf{life}} + W_{\mathsf{final}}\Delta_{\mathsf{bucket}}$. The pot must therefore hold working capital for in-flight submissions; a pot manager that lets it run dry stalls relaying and forces self-submission (Section 7).

B Epoch-Flooding Cost

A spend's anonymity set is at most the honest leaves in its input epoch (Section 6.3.9), so an attacker can shrink it by filling an epoch with its own leaves. A credit of value v drives at most $2v/M$ submissions, each appending two leaves, so at most about $4v/M$ leaves (Appendix A); occupying an epoch of capacity C therefore ties up value on the order of $M\,C$. Recovering that value is lossy: the attacker must redeem its assigned notes with a registered operator, and every redemption pays the operator/treasury revenue split, so each unit cycled back out is taxed by the split. Sustained flooding therefore costs a recurring fraction of $M\,C$ per epoch dominated, which makes deanonymization by flooding expensive at scale; the design lever is to size the product $M\,C$ against the value an adversary would spend to deanonymize.

C Nullifier Retention Bound

Nullifiers are garbage-collected after the fixed window $W_{\mathsf{nullset}} = \lceil T_{\mathsf{life}}/\Delta_{\mathsf{bucket}} \rceil + 3$ buckets (Section 6.3.7); this window is sized so the double-spend check stays both sound (a garbage-collected nullifier can never recur) and complete (both spends of a note land in still-active buckets). A note spent at h_{spend} was created no later than then, so $h_{\mathsf{exp}} \leq h_{\mathsf{create}} + T_{\mathsf{life}} + \Delta_{\mathsf{bucket}} \leq h_{\mathsf{spend}} + T_{\mathsf{life}} + \Delta_{\mathsf{bucket}}$ (the boundary alignment of Section 6.3.3 rounds up by less than Δ_{bucket}, and $h_{\mathsf{create}} \leq h_{\mathsf{spend}}$), and past h_{exp} the in-circuit freshness check rejects any spend. The freshness reference is prover-supplied within a small window $\delta < \Delta_{\mathsf{bucket}}$ of the current height (and never in the future), so a note stays spendable until at most δ past h_{exp}; the +3-bucket margin absorbs this grace and the boundary alignment, with slack $\Delta_{\mathsf{bucket}} - \delta$ at the garbage-collection boundary. Hence once the chain passes $h_{\mathsf{spend}} + T_{\mathsf{life}} + \Delta_{\mathsf{bucket}} + \delta$ no unexpired note can reproduce a nullifier filed in bucket B, which is why the contract may garbage-collect $\mathsf{nullset}[B]$ once $b(h_{\mathsf{now}}) \geq B + W_{\mathsf{nullset}}$. The same bound makes the check complete: both spends of one note fall within a span of at most $T_{\mathsf{life}} + \Delta_{\mathsf{bucket}} + \delta$, fewer than W_{nullset} buckets even at the boundary, so an earlier spend's bucket is still active when a later double-spend is attempted; and at most W_{nullset} buckets are ever active, so the per-spend check is bounded.

D Chain-Head Recency Tracking

Every state proof the client accepts must be tied to a recent finalized state root (Section 6.5). Rather than open a dedicated consensus connection that would reintroduce a metadata channel, the client tracks the head from attestations attached to the SMP message-delivery responses it already polls, either verifying finality itself in the sync-committee style [31] or taking the freshest state root consistent across its diverse servers as a recency floor [26]. The exact recency defense depends on the chosen chain: on a settlement layer with a light-client finality protocol (for example Ethereum mainnet, whose sync-committee attestations [31] the client can verify directly) it is cryptographic, whereas on a layer without one it reduces to the cross-server recency floor; we leave the choice open.

References

[1] A. Johnson et al. Avoiding the Man on the Wire: Improving Tor's Security with Trust-Aware Path Selection. https://arxiv.org/abs/1511.05453, 2015.

[2] Diaz, Halpin, Kiayias. Reward Sharing for Mixnets. https://nym.com/nym-cryptoecon-paper.pdf

[3] T. Ritter. All About Tor (presentation, v1.6). https://ritter.vg/p/tor-v1.6.pdf

[4] The Tor Project. Announcing Vanguards: Add-On Onion Services. https://blog.torproject.org/announcing-vanguards-add-onion-services/

[5] J. R. Douceur. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, 1999.

[7] SimpleX Chat. SimpleX: messaging and application platform (network overview). https://github.com/simplex-chat/simplexmq/blob/stable/protocol/overview-tjr.md

[8] Pew Research Center. A Week in the Life of Popular YouTube Channels. 2019. https://www.pewresearch.org/internet/2019/07/25/a-week-in-the-life-of-popular-youtube-channels/

[9] H. S. Xavier. The Web unpacked: a quantitative analysis of global Web usage. arXiv:2404.17095, 2024. https://arxiv.org/abs/2404.17095

[10] SimpleX Documentation on SMP, https://github.com/simplex-chat/simplexmq/blob/stable/protocol/simplex-messaging.md

[11] SimpleX Documentation on XFTP, https://github.com/simplex-chat/simplexmq/blob/stable/protocol/xftp.md

[12] SimpleX Documentation on the Post-Quantum Double Ratchet, https://github.com/simplex-chat/simplexmq/blob/stable/protocol/pqdr.md

[13] SimpleX Chat. SimpleX Channels: stateful information delivery and management. https://github.com/simplex-chat/simplex-chat/blob/stable/docs/protocol/channels-overview.md

[14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[15] Zcash. Zcash Protocol Specification. https://zips.z.cash/protocol/protocol.pdf, accessed 2025.

[16] Privacy Pools. Privacy Pools Documentation. https://docs.privacypools.com/, accessed 2025.

[17] M. Gillett and contributors. Epoch-based nullifier database. Polygon Miden GitHub discussion #356, 2022.

[18] S. Palladino. Global state Epochs. Aztec forum, 2024.

[19] SimpleX Chat. SimpleX Community Vouchers. https://simplex.chat/vouchers/, accessed 2025.

[20] FAT SOLUTIONS. Tongo Documentation (Protocol Introduction). https://docs.tongo.cash/protocol/introduction.html, accessed 2025.

[21] E. Ben-Sasson, I. Bentov, Y. Horesh, and S. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. IACR ePrint 2018/046, 2018. https://eprint.iacr.org/2018/046

[22] J. Groth. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT*, 2016. https://eprint.iacr.org/2016/260

[23] B. Whitehat, J. Baylina, and M. Bellés. Baby Jubjub Elliptic Curve. EIP-2494 (draft), 2020. https://eips.ethereum.org/EIPS/eip-2494

[24] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An Empirical Analysis of Anonymity in Zcash. USENIX Security Symposium, 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/kappos

[25] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. CRYPTO, 2022. https://eprint.iacr.org/2021/370

[26] SimpleX Chat. Trustless name resolution: forwarding Ethereum state proofs and chain-head synchronization. simplexmq issue 1816. https://github.com/simplex-chat/simplexmq/issues/1816

[27] J. Quesnelle. On the linkability of Zcash transactions. 2017. https://arxiv.org/abs/1712.01210

[28] X. Liu, S. Gao, T. Zheng, and B. Xiao. SnarkFold: Efficient SNARK Proof Aggregation from Split Incrementally Verifiable Computation. IACR ePrint 2023/1946, 2023. https://eprint.iacr.org/2023/1946

[29] D. Chaum. Blind Signatures for Untraceable Payments. In *CRYPTO*, 1982. https://www.chaum.com/publications/Chaum-blind-signatures.PDF

[30] Cashu. NUT-11: Pay-to-Pub-Key (P2PK). https://github.com/cashubtc/nuts/blob/main/11.md, accessed 2025.

[31] Ethereum. Altair Light Client – Sync Protocol (consensus specifications). https://github.com/ethereum/consensus-specs/blob/dev/specs/altair/light-client/sync-protocol.md, accessed 2025.

[32] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, and T. Hess. ERC-4337: Account Abstraction Using Alt Mempool. 2021. https://eips.ethereum.org/EIPS/eip-4337

[33] OpenGSN. Gas Station Network Documentation. https://docs.opengsn.org/, accessed 2025.

[34] Financial Crimes Enforcement Network (FinCEN). Bank Secrecy Act Regulations, General Definitions: Prepaid Access (closed-loop prepaid access and the prepaid-program exclusion), 31 C.F.R. § 1010.100. Final Rule, 76 Fed. Reg. 45403, 2011. https://www.ecfr.gov/current/title-31/section-1010.100

[35] United Kingdom. The Payment Services Regulations 2017, SI 2017/752, Schedule 1 paragraph 2(k) (limited network exclusion), with notification to the Financial Conduct Authority under regulation 38; implementing PSD2 (Directive (EU) 2015/2366) Article 3(k). https://www.legislation.gov.uk/uksi/2017/752