

Community Credits

A Private, Closed-Loop Payment Layer for the SimpleX Network

Working Draft

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

June 19, 2026

Abstract

SimpleX is a messaging network with no user profile identifiers, which gives it strong meta-data privacy but leaves open how a decentralized, censorship-resistant network pays the operators and developers it depends on: a general-purpose payment system would reintroduce the identification the network avoids. This draft presents *Community Credits*, a closed-loop payment layer that funds network capacity while preserving user privacy. Credits are prepaid, non-transferable, expiring claims on capacity, backed one-for-one by 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 contract-enforced revenue sharing. 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. We state which privacy goals the present design meets and which remain open.

Contents

1	The SimpleX Network	2
1.1	About SimpleX Network	2
2	Motivation	3
2.1	Collusion Resistance	3
2.2	Censorship and Coercion Resistance	3
2.3	Economic Viability	4
3	Channels and the Economic Model	4
4	Regulatory Considerations	4
5	Design Objectives and Threat Model	5
5.1	Functional and Economic Objectives	5
5.2	Privacy Goals	5
5.3	Threat Model	6
6	Private Blockchain Access	7

7	Community Credits	8
7.1	Background and Related Work	9
7.1.1	Note-Based Privacy Systems	9
7.1.2	Homomorphic Ciphertext Updates	9
7.1.3	zkSNARKs (Groth16)	9
7.2	Design Overview	10
7.2.1	Roles	10
7.2.2	Dual Public/Private Ledger	10
7.2.3	Threat Model	11
7.3	Protocol	11
7.3.1	Keys and Voucher Format	11
7.3.2	On-chain State	13
7.3.3	Voucher Lifecycle	13
7.3.4	Nullifier Bucketing and Garbage Collection	18
7.3.5	Expiration and Reimbursement	18
7.3.6	Epoch-Based Commitment Trees	18
7.4	Smart Contract Design	19
7.4.1	Proof Verification and Aggregation	19
7.4.2	Token Interfaces and Restricted Transfers	19
7.4.3	Roles and Flows	19
8	Analysis and Open Questions	20
9	Conclusion	21

1 The SimpleX Network

1.1 About SimpleX Network

SimpleX is a privacy- and security-focused messaging network with no user profile identifiers — no accounts, phone numbers, usernames, or even random numbers assigned 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 carries data over the public Internet through nodes (SMP relays) that accept, 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”) hides 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 carry 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 whose 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, censorship-resistant network must pay its operators, and it cannot do so through a general-purpose payment rail without destroying the metadata privacy SimpleX is built 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, which — depending on the system — takes a majority of the nodes, as in proof-of-work, or at least a third of them, since Byzantine-fault-tolerant systems require two-thirds honest [5, 6]. Deanonimization needs far less: an adversary who operates relays at both ends of a user’s path can link that user’s connections by correlating timing and volume, using only a small fraction of the relays. Open-participation networks such as Tor are exposed here. Tor is, in fact, permissioned — a small set of directory authorities controls which relays appear in its consensus — but participation is pseudonymous, anyone may run a relay, and a user has no assurance about who operates the relays on their path. Ritter estimates that operating on the order of 2% of relays already correlates a meaningful fraction of streams [3]; Tor’s Vanguard defense raised the cost to “weeks or even months, even if they run 5% of the network” [4]. Economic barriers such as deposits or staking raise the price of running relays at scale [2], but none of this puts 5% of the network beyond a well-funded — let alone state-level — 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. 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.

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 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), ensuring that neither the queried state nor the querier’s identity is exposed to any single party.

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 7) 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 layer.

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 layer 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; Community Credits carry the payment.

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 both 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 never takes custody of user funds or keys, never issues stablecoin, and never exchanges between fiat and crypto. In the United States the system is structured to fit the closed-loop prepaid-access exemption: every payee is identified to buyers in advance through the public operator registry, and per-buyer purchases are limited to no more than \$2,000 per day, enforced in the app in the same way prepaid telecom cards apply daily limits. In the United Kingdom it is structured to fit the limited-network exemption: an admission-controlled list of Network Operators bound by a signed Operator Deed, redeemable only through the SimpleX

interface rather than an open marketplace, with prior notification to the Financial Conduct Authority. The analysis turns on the recipients of funds being identified and confined to a known list — not on the payees being regulated — while the design conceals only the payer and the payment’s link to a community.

5 Design Objectives and Threat Model

Community Credits are a payment rail: the design defines how value moves through the network. The objectives below, and the privacy goals in particular, shaped the draft design described in the following sections.

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 or returned to the treasury, and expiring after a fixed lifetime.
- **Arbitrary amounts, with assignment and change.** A credit may hold any value. 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 can neither be re-assigned nor transferred once assigned.
- **Solvency by construction.** The face value of all credits in circulation 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 split of revenue between operators and the network is enforced by the contract, not by trust.
- **Holder recovery.** A user or community that loses its device can recover its credit balance from a single backed-up secret, without assistance from any operator or other party.
- **Mobile-provable redemption.** The zero-knowledge proof a community produces to redeem a credit must be generated on a mid-range mobile phone within a few seconds.

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. Buyers have no persistent on-chain identity — each purchase uses a fresh address — and the per-buyer daily limit is applied at the point of sale.

- **Assignment is private.** Assigning a credit to a community reveals neither the amount assigned nor 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 unobservable as an event.** An individual redemption reveals neither its amount, nor which operator was paid, nor which expiration cohort — and therefore which purchase period — the credit belonged to. Redemptions cannot be linked to one another, to an assignment, or to a purchase.
- **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 both the community and the donor. Concealing redemptions as events, and revealing only aggregate operator earnings, removes that correlation.

Status in the draft design. Assignment hides both the amount assigned and the recipient community. Both assignment and redemption, however, publish the note’s expiration height. This does not pin the exact purchase: the client aligns each credit’s expiration to a coarse boundary such as a quarter, so a nominal twelve-month credit may expire anywhere in a nine-to-fifteen-month window and a whole range of purchases lands on the same height. The published value therefore reveals only the expiration bucket, not the purchase. Redemption additionally reveals the amount and the operator, so individual redemptions are observable as events. Unlinkability to the purchase is nonetheless preserved — the spend reveals no commitment. The only values that could tie a redemption back to a purchase are its amount and its expiration, and both are kept low-cardinality: the amount by spending notes in a small set of fixed denominations, the expiration by quarter-bucketing. Correlation over them therefore yields at most a coarse cohort, never a link to a specific purchase. The operator is chosen at redemption and carries no such information. Concealing these values entirely — so that only aggregate operator withdrawals are public and not even the bucket is disclosed — would require protocol changes the draft does not adopt; Section 8 sets out the direction.

5.3 Threat Model

Adversary. We assume an adversary that observes all on-chain activity and all network traffic reaching the blockchain, controls part of the communication network, and may operate or corrupt some Network Operators and indexer nodes. The adversary may itself act as a buyer, a community, or an operator. Indexer and registry nodes are honest-but-curious for privacy: they follow the protocol but attempt to learn what they can. We assume the cryptographic primitives are secure and that users keep their device secrets safe.

Operators are publicly identified. The set of Network Operators is a public, admission-controlled list: each operator is identified, bound by the Operator Deed, and named in the registry,

and the contract permits only registered operators to withdraw. The closed-loop regulatory posture and the network’s trust model both require this. Concealing *which* registered operator a given redemption pays is a privacy goal; the operator identities themselves are public.

Custody and solvency. The contract holds the stablecoin and issues credits only against deposited backing, so the face value in circulation never exceeds what has been deposited, and no party — operator or otherwise — can move a user’s redeemable funds. Expired value is reclaimed only in aggregate and only on a verifiable proof of the per-bucket totals, never for an individual credit.

What is not protected. We do not conceal the total stablecoin held by the pool, each operator’s aggregate earnings, coarse timing, or the per-cohort totals disclosed when an expired bucket is reclaimed. Nor do we defend against an adversary who already knows by external means that a particular person controls a particular credit and who watches that person’s own device or network connection; such targeted attacks are out of scope here, as they are for the underlying SimpleX transport.

6 Private Blockchain Access

Several components of the SimpleX network rely on smart contract state as a single source of truth (the operator registry and the Community Credits contract). Reading this state must not leak metadata about who is querying or what they are querying.

A conventional light client participates in the blockchain’s peer-to-peer network to fetch headers and state proofs. This reveals the client’s IP address and the specific contract storage slots it requests, allowing observers, including blockchain full nodes, to correlate queries with network identities.

SimpleX replaces the networking layer of the light client with its own onion-routed relay infrastructure while retaining the cryptographic verification. Concretely, a SimpleX client reads blockchain state through a two-hop path via SMP relays using private routing:

1. The client encrypts its query (contract address, storage slot or call) for an *indexer node*, a SimpleX node that maintains a local copy of the relevant contract state and can produce state proofs. The encrypted query is sent through a forwarding relay chosen by the client.
2. The forwarding relay sees the client’s network address but cannot read the query (it is encrypted for the indexer). The indexer receives the query and returns the result with a state proof, but does not learn who sent it (it sees only the relay’s address).
3. The client verifies the state proof locally using light-client cryptography (block headers, Merkle–Patricia proofs or equivalent). This verification requires no network interaction beyond the single round-trip above.

This design provides the same trust-minimization as a conventional light client (the client does not trust the indexer’s answer, only the cryptographic proof) while ensuring that no single party learns both the querier’s identity and the query content. Indexer nodes can be operated by SMP relay operators (discoverable via the SORC, Definition 1) or by dedicated infrastructure providers.

7 Community Credits

Community Credits are a means of paying for utility in the SimpleX network anonymously. More specifically, credits shall allow end users to fund group channels so they can pay super-peers and XFTP relay operators on behalf of the group

Operationally, vouchers behave like prepaid credit. The SimpleX Network users can purchase the vouchers via dApp for stablecoin receiving *private voucher notes* to use inside the app. [19] At a later point, support for in-app payments for vouchers can be added. Only registered network operators and app developers can withdraw the funds.

Out-of-band note delivery (no on-chain encrypted notes). Unlike many privacy-pool designs where an *encrypted note ciphertext* is published on-chain so wallets can discover incoming notes by scanning, Community Vouchers deliver notes *out-of-band over SimpleX*: the issuer sends the note plaintext (or an encrypted payload from which the wallet can recover it) directly to the user via SimpleX messages. The on-chain contract stores only commitments, nullifiers, and aggregate bucket state — *not* encrypted notes in contract storage. Storing encrypted note ciphertext on-chain (e.g., as calldata/event data for self-contained archival) is an optional extension, but the simplified flow (assignment to communities followed by redemption, with no general-purpose transfers) makes out-of-band delivery lighter and operationally simpler.

Vouchers are non-tradable utility credits tied to SimpleX Network server capacity. They have (i) restricted redemption to permissioned operators/treasury, and (ii) a fixed lifetime in blocks (e.g., ~12 months), after which individual notes become unspendable and the funds for expired credits can be reclaimed only in aggregate by SimpleX Network Consortium. [19]

Design requirements.

- **Privacy.** Purchases (in-app or off-chain) must not be linkable to the communities that redeem the vouchers. Assignment of a voucher to a community must not reveal the purchase transaction.
- **Assignable vouchers.** Vouchers support any amount. A purchaser can assign a voucher (or a portion of it, with change) to a community. Only the assigned community can redeem the voucher with operators.
- **Restricted use.** Vouchers are closed-loop credits for SimpleX capacity, not general-purpose payment tokens. Partial redemptions by the same community are unrestricted. Vouchers cannot be re-assigned or transferred after assignment.
- **Expiring liabilities.** Notes have lifetime T_{life} ; after expiry they cannot be redeemed, and residual value is reclaimed only in aggregate.
- **Zero-trust settlement.** Stablecoins stay locked in contracts during redemption; operators accrue internal claims and withdraw later under on-chain revenue-share agreements.

Technically, Community Vouchers are a permissioned privacy pool: private notes and nullifiers (Zcash-style), [15] constrained egress and compliance boundaries (Privacy Pools-style), [16] and epoch-based commitment structures (Aztec/Miden-style) to keep state bounded. [17, 18]

Contributions. This section proposes:

- A compact protocol for a permissioned privacy pool with fixed-lifetime notes for community payments, supporting private assignment of vouchers to communities and subsequent redemption by the assigned community.
- A dual-ledger model: a *public* stablecoin/claims ledger decoupled from a *private* note/nullifier ledger, enabling efficient liquidity management and revenue sharing under permissioned operators.
- An on-chain state layout with epoch-based (incremental) commitment trees, expiration-bucketed nullifiers with garbage collection, and bucketed liabilities tracked via additively homomorphic encryption (AHE) without centralized counters.
- A smart contract architecture that verifies Groth16 zkSNARK proofs for *voucher creation, assignment, and redemption* and supports relayed fee sponsorship without privacy leakage.

7.1 Background and Related Work

7.1.1 Note-Based Privacy Systems

Community Vouchers 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 Vouchers take a narrower closed-loop form: withdrawals (stablecoin egress) are operator/network-only. [16]

To bound on-chain state, we organize commitments into per-epoch trees and keep a compact structure committing to historical roots, as in Aztec/Miden-style epoch designs and related accumulator techniques. [17, 18]

7.1.2 Homomorphic Ciphertext Updates

Zether illustrates how contracts can maintain encrypted balances with an additively homomorphic encryption scheme while users provide ZK proofs that link plaintext updates to ciphertext updates. [20] We reuse the same high-level pattern, but for *bucket-level* liabilities (spent totals per expiration bucket).

7.1.3 zkSNARKs (Groth16)

Community Vouchers require users (communities) to generate zero-knowledge proofs on their mobile devices when redeeming vouchers. This constraint is decisive for the choice of proof system.

zkSTARKs offer attractive properties — transparent setup (no trusted ceremony), post-quantum security, and efficient proof aggregation [22]. However, STARK proof generation requires 4–8 GB of RAM and 5–15 seconds even on flagship mobile phones. On mid-range devices (under 3 GB RAM, common in developing markets), STARK provers crash approximately 30% of the time. This makes STARKs impractical for a user-facing mobile application.

We target Groth16 zkSNARKs [23] instead. Groth16 provers require under 1.5 GB of RAM and complete in 0.1–3 seconds on mobile devices, with negligible crash rates. Groth16 proofs are 128 bytes (constant size), compared to 50–200 KB for STARKs, reducing on-chain verification cost. On-chain Groth16 verification requires approximately 200K gas in Solidity — well within the budget for an EVM-compatible L2.

Two proving paths: browser WASM today, native tomorrow. The proving time budget above admits two concrete implementations of the Groth16 prover, both supported by the circuit and key choices in this paper:

- *Browser WASM (snarkjs)*: a portable JavaScript / WebAssembly prover that runs inside any mobile in-app dapp browser. Acceptable for a public web-deployed dapp where the user has no native app installed. With circuits at the $\sim 5K$ -constraint scale used here (Section 7.3.3 and Section 7.3.3), snarkjs delivers ~ 1 s median assign/redeem proving on mid-range phones in current measurements.
- *Native (rapidsnark / mopro)* [25, 26]: a native-compiled Groth16 prover invoked via FFI from an iOS or Android app. Reaches the lower end of the 0.1–3 s band (~ 200 –500 ms) on the same hardware, primarily by exploiting SIMD-tuned multi-scalar multiplication that browser WASM cannot match.

The same circuits and proving keys are consumed by both paths. As the chat component of the protocol matures from a public web dapp into the SimpleX Chat native mobile app, the proving path is expected to migrate from WASM to native without protocol-level changes.

Groth16 requires a per-circuit trusted setup ceremony: a structured reference string (SRS) must be generated for each circuit via a multi-party computation (MPC). While this is an operational cost compared to transparent-setup systems, the ceremony is a one-time event per circuit version and can be conducted with public verifiability. Given the mobile proving constraint, this trade-off is acceptable.

7.2 Design Overview

7.2.1 Roles

Users (purchasers) Hold private voucher notes delivered out-of-band over SimpleX. Assign vouchers to communities by generating zkSNARK proofs.

Communities (redeemers) Receive assigned vouchers from purchasers and generate zkSNARK proofs to redeem them with operators.

Operators Run SimpleX messaging servers. They are the only entities (besides treasury) allowed to withdraw stablecoins, and they do so only by withdrawing previously accrued on-chain claims.

Voucher contract Sells vouchers on-chain for stablecoin, holds stablecoin reserves, maintains commitment trees and nullifier sets, verifies *creation*, *assignment*, and *redemption* proofs, enforces permissioning, and applies revenue sharing. These functions can be split between several contracts.

SimpleX nodes/indexers Follow voucher events and serve Merkle authentication paths to clients via onion-routed SimpleX protocol requests.

7.2.2 Dual Public/Private Ledger

Community Vouchers maintain two coupled ledgers:

- **Public ledger (stablecoin + claims)**. On-chain, the contract tracks stablecoin reserves via explicit accounting, operator claims (credits), network claims, and revenue-share agreements. These determine who can withdraw stablecoins and how much.

- **Private ledger (notes + nullifiers).** User balances stored as private notes (commitments). Spending publishes nullifiers and updates the commitment tree by appending a new (change) note, without revealing the note identity.

This separation decouples (i) *funding* from *minting* and (ii) *redemption* from *withdrawal*. Permitted operators make this safe: operators are the only stablecoin withdrawers, and all withdrawals are mediated by on-chain agreements. The result is clean revenue sharing (applied at voucher redemption time).

7.2.3 Threat Model

Adversaries can observe all on-chain activity, control parts of the communication network, and corrupt some operators/indexers. Users are assumed to keep their device keys safe. We hide *linkability* between voucher purchases, assignment to communities, and later redemptions; we do not hide coarse timing or global aggregates needed for transparency.

We assume note delivery occurs out-of-band via SimpleX messaging. To reduce operational errors and enable verifiable receipts despite out-of-band delivery, voucher creation includes a zk-SNARK proof binding the on-chain commitment to a well-formed note and (optionally) to a commitment of the encrypted note payload that was sent to the user.

7.3 Protocol

7.3.1 Keys and Voucher Format

Each user generates a spending key sk , a uniformly random field element. The corresponding "public key" is the Poseidon-based hash $pk = \text{Poseidon}(sk)$. We deliberately avoid an elliptic-curve-based keypair here (cf. earlier drafts that used Baby Jubjub $pk = sk \cdot G$): an in-circuit scalar multiplication on Baby Jubjub costs roughly 10K constraints, which dominates the assign and redeem circuits. A Poseidon-based ownership hash collapses that to a single ~ 150 -constraint Poseidon evaluation, bringing the assign and redeem circuits down to $\sim 5K$ constraints and keeping browser-WASM proving on mid-range mobile devices within the proving-time budget stated in Section 7.1.3 (0.1–3 s). The security argument relies on collision-resistance of Poseidon and soundness of Groth16, both of which are sufficient for the closed-loop permitted voucher protocol described here; we never need EC-based digital signatures because note authorization is purely an in-circuit knowledge-of-preimage statement.

A voucher note is:

$$\text{note} = (v, h_{\text{exp}}, pk, r, \text{assigned}, h_{\text{redeemer}}),$$

where v is value, h_{exp} is the expiration height, r is commitment randomness, $\text{assigned} \in \{0, 1\}$ indicates whether the voucher has been assigned to a redeemer, and $h_{\text{redeemer}} = H(\text{id}_{\text{redeemer}})$ is a hash commitment to the redeemer's identity (set to 0 when $\text{assigned} = 0$). The on-chain commitment is

$$\text{cm} = \text{Com}(\text{note}).$$

The nullifier is

$$\text{nf} = \text{Poseidon}(sk, \text{cm}),$$

revealed only when redeeming the voucher. (Compared to a separate nullifier key $nk = \text{Poseidon}(sk)$ followed by $\text{nf} = \text{PRF}_{nk}(\text{cm})$, this single-hash form is equivalent under the same Poseidon assumptions and saves one in-circuit hash.)

Out-of-band note payloads (SimpleX delivery). The contract does not store encrypted notes. Instead, the dApp provides each newly minted note to the recipient. This is sufficient because (i) notes support only assignment to communities and redemption (no general-purpose transfers), and (ii) recipients do not need to discover inbound notes by scanning chain ciphertexts.

Optional note payload encryption commitment. Although SimpleX provides end-to-end encrypted transport, we model issuance as producing a well-defined *note payload ciphertext* to (a) decouple note confidentiality from transport details and (b) enable a zkSNARK *creation proof* that the note was constructed and encrypted correctly. Concretely, a user provides the issuer an ephemeral note-delivery public key pk_{del} (distinct from (sk, pk)). The issuer forms:

$$\text{ct} = \text{Enc}_{\text{pk}_{\text{del}}}^{\text{note}}(\text{note}; \eta),$$

for some randomized public-key encryption / ECIES-style scheme Enc^{note} and nonce η . The issuer computes compact commitments:

$$h_{\text{ct}} = H(\text{ct}), \quad h_{\text{del}} = H(\text{pk}_{\text{del}}),$$

for a hash H suitable for the zkSNARK circuit (e.g., Poseidon). The ciphertext ct is sent over SimpleX (out-of-band); the on-chain transaction need only reference h_{ct} and h_{del} .

Additively homomorphic encryption (AHE) on Baby Jubjub. The issuer generates an AHE keypair $(\text{pk}_I, \text{sk}_I)$ for encrypting bucket-level spent totals. We instantiate ElGamal-in-the-exponent over Baby Jubjub [24], a twisted Edwards curve embedded in the BN254 scalar field. Baby Jubjub is the standard curve for in-circuit elliptic curve operations in Groth16 (BN254-based) proof systems, enabling efficient proof generation for ciphertext update statements. On-chain verification uses the BN254 precompile available on EVM-compatible chains.

Write the curve group multiplicatively for notational convenience. Let $G = \langle g \rangle$ be the prime-order subgroup (in implementation, g^x corresponds to scalar multiplication $x \cdot G$ on the curve). Let $\text{pk}_I = g^{\text{sk}_I}$. Encryption of an integer x with randomness (nonce) ρ is:

$$\text{Enc}_{\text{pk}_I}(x; \rho) = (C_L, C_R) = (g^x \cdot \text{pk}_I^\rho, g^\rho), \quad \rho \xleftarrow{\$} \mathbb{Z}_q.$$

Ciphertext addition is component-wise multiplication:

$$(C_L, C_R) \oplus (C'_L, C'_R) = (C_L \cdot C'_L, C_R \cdot C'_R),$$

corresponding to plaintext addition in the exponent.

Decryption and bounded discrete log. Decryption yields:

$$g^x = C_L / C_R^{\text{sk}_I}.$$

Recovering x from g^x requires a discrete log. Here, x is bounded by policy (bucket-level totals), so the issuer can recover x via bounded search (tables or baby-step/giant-step), as in Zether-style balance recovery. [20]

7.3.2 On-chain State

The voucher contract maintains:

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

deposited, withdrawn,

satisfying:

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

- **Available mint capacity.** A scalar

available_mint,

tracking how much additional voucher face value the issuer may mint without additional backing. This variable *persists across epochs* (carryover) and evolves as:

$$\text{fundPool}(a) : \text{available_mint} \leftarrow \text{available_mint} + a,$$

$$\text{createVoucher}(\cdot, v, \cdot) : \text{available_mint} \leftarrow \text{available_mint} - v.$$

If expired liabilities are reclaimed to an issuer-defined sink that remains inside the pool (i.e., value is not withdrawn as stablecoin), reclaimed amounts may re-enter mintable capacity by increasing `available_mint` (Section 7.3.5).

- **Operator registry.** Authorized operators with payout addresses 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 compact structure committing to historical roots (Section 7.3.6).
- **Expiration-bucketed nullifier sets.** Mapping from expiration bucket e to sets $\text{nullset}[e]$ of used nullifiers.
- **Liability buckets.** For each expiration bucket e :

minted $[e]$, $C_{\text{spent}}[e]$,

where $\text{minted}[e]$ is the plaintext total minted face value whose notes expire in e , and $C_{\text{spent}}[e]$ is an AHE ciphertext of the total redeemed/cancelled from that bucket.

- **Operator balances.** For each operator o , an internal balance $\text{credit}[o]$ (redeemed value not yet withdrawn). Revenue sharing is applied when o withdraws (Section 7.4.3).

7.3.3 Voucher Lifecycle

Let h be the current block height. We divide block heights into *expiration buckets* of size Δ_{bucket} blocks (e.g., month-like). Let $b(h)$ be the bucket index containing h . A note with expiration height h_{exp} belongs to bucket $e = b(h_{\text{exp}})$.

Issuer Funding and Mint Capacity Issuer accounts fund the pool via:

$$\text{fundPool}(a),$$

which transfers a units of S into the contract. The contract updates:

$$\text{deposited} \leftarrow \text{deposited} + a, \quad \text{available_mint} \leftarrow \text{available_mint} + a.$$

Decoupling funding and minting. Unlike typical privacy pools where a user deposit is atomically the note mint, here funding is centralized (issuer-only) and minting can lag funding. The key solvency guard is that every minted voucher reduces `available_mint`, so minted liabilities never exceed funded capacity at the contract level.

Off-Chain Voucher Creation Upon purchase confirmation, the issuer:

1. sets $h_{\text{exp}} = h_{\text{create}} + T_{\text{life}}$;
2. constructs $\text{note} = (v, h_{\text{exp}}, \text{pk}, r, 0, 0)$ and commitment $\text{cm} = \text{Com}(\text{note})$ (unassigned);
3. encrypts the note payload for delivery using the user's ephemeral delivery key pk_{del} , producing $\text{ct} = \text{Enc}_{\text{pk}_{\text{del}}}^{\text{note}}(\text{note}; \eta)$ and commitments $h_{\text{ct}} = H(\text{ct})$, $h_{\text{del}} = H(\text{pk}_{\text{del}})$;
4. sends the note payload ciphertext ct (or wallet decryption material) to the user via SimpleX (out-of-band).

On-Chain Voucher Creation Voucher creation also includes an issuer-generated zkSNARK proof to ensure the note was constructed and encrypted correctly. The issuer calls:

$$\text{createVoucher}(\text{cm}, v, h_{\text{exp}}, h_{\text{ct}}, h_{\text{del}}, \pi_{\text{create}}).$$

Mint statement (proved in zkSNARK). The prover (issuer) shows knowledge of $(\text{note}, \text{pk}_{\text{del}}, \eta)$ such that:

- note well-formedness: $\text{note} = (v, h_{\text{exp}}, \text{pk}, r, 0, 0)$ and $\text{cm} = \text{Com}(\text{note})$ (unassigned);
- delivery-key commitment: $h_{\text{del}} = H(\text{pk}_{\text{del}})$;
- payload encryption correctness: $\text{ct} = \text{Enc}_{\text{pk}_{\text{del}}}^{\text{note}}(\text{note}; \eta)$ and $h_{\text{ct}} = H(\text{ct})$;
- domain separation / replay resistance: the statement commits to the chain id and voucher contract address.

Public inputs include $(\text{cm}, v, h_{\text{exp}}, h_{\text{ct}}, h_{\text{del}})$ (and the domain separators).

Contract checks. The contract checks:

1. caller is an authorized issuer;
2. h_{exp} is consistent with T_{life} within tolerance;
3. **available mint capacity:** $v \leq \text{available_mint}$;
4. the current epoch tree has space for another leaf;

5. verifies the zkSNARK creation proof π_{create} (Section 7.4.1).

If checks pass, the contract:

- appends cm to the current epoch incremental Merkle tree (updates frontier/root);
- updates $\text{available_mint} \leftarrow \text{available_mint} - v$;
- computes $e = b(h_{\text{exp}})$ and updates $\text{minted}[e] \leftarrow \text{minted}[e] + v$;
- initializes $C_{\text{spent}}[e] = \text{Enc}_{\text{pk}_I}(0)$ if bucket e is new;
- emits a **VoucherCreated** event containing $(\text{cm}, v, h_{\text{exp}}, h_{\text{ct}}, h_{\text{del}})$ so the recipient can verify the out-of-band payload against an on-chain receipt, without storing encrypted notes on-chain.

Optional on-chain ciphertext publication. A deployment may additionally publish ct on-chain (e.g., as calldata or event data) for self-contained archival and delayed retrieval. This increases calldata/on-chain data, so the default design prefers SimpleX out-of-band delivery; the same creation proof can be adapted to bind directly to an on-chain ct .

Inclusion Proofs via SimpleX Nodes Indexer nodes follow voucher events and maintain epoch trees. A client requests a Merkle path via onion-routed SimpleX messages:

`getPath(epoch, root, cm) -> (exists, index, path).`

Incremental Merkle tree witnesses. Commitment trees are maintained incrementally (append-only) with a *frontier* representation, mirroring Zcash note commitment trees and their incremental witnesses: wallets can update their witness as new commitments are appended, without revealing which note they own. [15]

Note payloads are not served from chain. Note payloads themselves are delivered out-of-band over SimpleX during voucher creation; indexers are used for membership paths (witness maintenance) and, optionally, for relaying assignment and redemption payloads.

Assignment Assignment transfers a voucher (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 redeemer (destination) and one unassigned (change) returned to the purchaser. Both outputs preserve the original expiration height.

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

$$\begin{aligned} \text{note}_{\text{dest}} &= (v_{\text{dest}}, h_{\text{exp}}, \text{pk}_{\text{dest}}, r_{\text{dest}}, 1, h_{\text{redeemer}}), \\ \text{note}_{\text{change}} &= (v_{\text{change}}, h_{\text{exp}}, \text{pk}, r_{\text{change}}, 0, 0), \end{aligned}$$

with commitments cm_{dest} and $\text{cm}_{\text{change}}$, satisfying $v_{\text{dest}} + v_{\text{change}} = v$. Full assignment uses $v_{\text{change}} = 0$.

Assign statement (proved in zkSNARK). The prover (purchaser) shows knowledge of $(\text{sk}, \text{note}_{\text{in}}, \text{path}, r_{\text{dest}}, r)$ such that:

- $\text{cm} = \text{Com}(\text{note}_{\text{in}})$ and $\text{cm} \in T_E$ with root R_E ;
- ownership: $\text{pk} = \text{Poseidon}(\text{sk})$;
- nullifier correctness: $\text{nf} = \text{Poseidon}(\text{sk}, \text{cm})$;
- input is unassigned: $\text{assigned} = 0$;
- the input note has not expired: h_{exp} is greater than the current block height;
- amount conservation: $v_{\text{dest}} + v_{\text{change}} = v$ with $v_{\text{dest}} > 0$;
- output correctness: $\text{cm}_{\text{dest}} = \text{Com}(\text{note}_{\text{dest}})$ with $\text{assigned} = 1$ and $h_{\text{redeemer}} = H(\text{id}_{\text{redeemer}})$;
- change correctness: $\text{cm}_{\text{change}} = \text{Com}(\text{note}_{\text{change}})$ with $\text{assigned} = 0$;
- expiration preservation: both outputs have the same h_{exp} as the input;
- domain separation / replay resistance: the statement commits to the chain id and voucher contract address.

Public inputs include $(E, R_E, \text{nf}, h_{\text{exp}}, \text{cm}_{\text{dest}}, \text{cm}_{\text{change}})$.

Contract checks for assignment. The contract:

1. verifies (E, R_E) against the epochs structure;
2. checks freshness: current height $h_{\text{now}} \leq h_{\text{exp}}$;
3. computes $e = b(h_{\text{exp}})$ and checks $\text{nf} \notin \text{nullset}[e]$;
4. verifies the zkSNARK assignment proof π_{assign} ;
5. records the nullifier: insert nf into $\text{nullset}[e]$;
6. appends cm_{dest} and $\text{cm}_{\text{change}}$ to the current epoch commitment tree.

Assignment does not move stablecoins or update liability buckets — it only restructures note ownership within the private ledger. The purchaser delivers the destination note payload to the community out-of-band over SimpleX.

Redemption (Canonical Spend Statement) A redemption consumes one *assigned* input note and produces one output (change) note to the same owner (community). Full redemption is $v_{\text{change}} = 0$; partial redemption has $0 < v_{\text{change}} < v$; cancellation is a redemption to a designated treasury sink.

Let the input note be $(v, h_{\text{exp}}, \text{pk}, r, 1, h_{\text{redeemer}})$ with commitment cm included in epoch tree T_E under root R_E . Let the output note be $\text{note}' = (v_{\text{change}}, h_{\text{exp}}, \text{pk}, r', 1, h_{\text{redeemer}})$ with commitment cm' (same owner key and redeemer). Define:

$$v_{\text{out}} = v - v_{\text{change}} \geq 0.$$

Let $e = b(h_{\text{exp}})$ and let $(C_{\text{spent}}^{\text{old}}[e])$ be the current on-chain ciphertext for bucket e .

Spend statement (proved in zkSNARK). The prover (community) shows knowledge of $(\text{sk}, \text{note}, \text{path}, r', \text{id}_{\text{redeemer}})$ such that:

- $\text{cm} = \text{Com}(v, h_{\text{exp}}, \text{pk}, r, 1, h_{\text{redeemer}})$ and $\text{cm} \in T_E$ with root R_E ;
- ownership: $\text{pk} = \text{Poseidon}(\text{sk})$;
- assignment: $\text{assigned} = 1$ and $h_{\text{redeemer}} = H(\text{id}_{\text{redeemer}})$;
- nullifier correctness: $\text{nf} = \text{Poseidon}(\text{sk}, \text{cm})$;
- output correctness: $\text{cm}' = \text{Com}(v_{\text{change}}, h_{\text{exp}}, \text{pk}, r', 1, h_{\text{redeemer}})$ and $v_{\text{out}} + v_{\text{change}} = v$;
- ciphertext update correctness: there exists user-chosen randomness ρ and

$$C_{\Delta} = \text{Enc}_{\text{pk}_I}(v_{\text{out}}; \rho),$$

- domain separation / replay resistance: the statement commits to the chain id and voucher contract address (and, if desired, to a chosen relayer/caller address), following Tongo-style replay protections. [21]

Public inputs include $(E, R_E, \text{nf}, h_{\text{exp}}, v_{\text{out}}, \text{cm}', C_{\Delta})$.

Contract checks and state updates. Given a spend targeting either an operator o (redemption) or treasury sink (cancellation), the contract:

1. verifies (E, R_E) against the epochs structure (Section 7.3.6);
2. checks freshness: current height $h_{\text{now}} \leq h_{\text{exp}}$;
3. computes $e = b(h_{\text{exp}})$ and checks $\text{nf} \notin \text{nullset}[e]$;
4. if recipient is an operator, checks o is registered and active in the operator registry;
5. verifies the zkSNARK proof (Section 7.4.1);
6. records the nullifier: insert nf into $\text{nullset}[e]$;
7. updates the liability ciphertext:

$$C_{\text{spent}}[e] \leftarrow C_{\text{spent}}[e] \oplus C_{\Delta};$$

8. appends cm' to the current epoch commitment tree (incremental update);
9. credits the recipient:

$$\text{credit}[o] \leftarrow \text{credit}[o] + v_{\text{out}}$$

(or $\text{credit}[\text{treasury}]$ for cancellation).

No stablecoins leave the pool during redemption; stablecoin egress happens only at withdrawal, where revenue sharing is enforced (Section 7.4.3).

7.3.4 Nullifier Bucketing and Garbage Collection

Nullifiers are stored by the expiration bucket $e = b(h_{\text{exp}})$. Once the chain passes h_{exp} , spends fail the expiry check, so old nullifiers need only be retained through a safety window.

Let $e_{\text{now}} = b(h_{\text{now}})$. The contract allows reclamation for bucket e only once $e_{\text{now}} \geq e + 2$, and after reclamation it can garbage-collect both $\text{nullset}[e]$ and the associated liability bucket state.

7.3.5 Expiration and Reimbursement

For each bucket e , the contract stores plaintext $\text{minted}[e]$ and ciphertext $C_{\text{spent}}[e]$. The issuer decrypts off-chain to obtain $\text{spent}[e]$ and proves correctness of the plaintext decryption to the contract (verifiable decryption), without revealing per-user voucher identities. [20]

Define:

$$L[e] = \text{minted}[e] - \text{spent}[e],$$

the unredeemed (expired) liability for bucket e .

The issuer calls:

$$\text{reclaimExpired}(e, s_e, \pi_e),$$

where s_e is the claimed plaintext spent total for e and π_e proves $C_{\text{spent}}[e]$ decrypts to s_e under pk_I . The contract:

- checks $e_{\text{now}} \geq e + 2$;
- verifies π_e and computes $v_{\text{reclaim}} = \text{minted}[e] - s_e$;
- applies policy to v_{reclaim} :
 - *withdrawal mode*: transfer v_{reclaim} (or fraction) to treasury/reward contracts and update $\text{withdrawn} \leftarrow \text{withdrawn} + v_{\text{reclaim}}$; OR
 - *re-mint mode*: keep stablecoins in the pool (no transfer) and increase mintable capacity:

$$\text{available_mint} \leftarrow \text{available_mint} + v_{\text{reclaim}}.$$

This corresponds to reclaiming expired liabilities to an issuer-defined sink that re-enters minting capacity.

- sets $\text{minted}[e] \leftarrow s_e$ and resets $C_{\text{spent}}[e]$ appropriately so residual $L[e] = 0$ going forward;
- garbage-collects $\text{nullset}[e]$ and bucket storage if desired.

7.3.6 Epoch-Based Commitment Trees

Commitments are organized into per-epoch append-only Merkle trees T_E . Only the current epoch tree is writable. When epoch E ends, its final root R_E is frozen and appended as a leaf in an *epochs structure* committing to historical roots.

Incremental tree maintenance. The contract stores the current epoch root and a frontier (incremental Merkle representation) to support efficient on-chain appends, mirroring Zcash-style incremental commitment trees. [15]

To spend a note created in epoch E , the zkSNARK proves membership in T_E under R_E , and the contract verifies (E, R_E) is included in the epochs structure.

7.4 Smart Contract Design

We outline a deployment on an EVM-compatible L2 rollup settling to Ethereum mainnet. The voucher contract is implemented in Solidity, with Groth16 proof verification via the BN254 elliptic curve precompile (EIP-196/197) available on all EVM-compatible chains.

7.4.1 Proof Verification and Aggregation

Community Vouchers use Groth16 zkSNARK proofs for:

- **Voucher creation (issuer proof):** to attest that the commitment corresponds to a well-formed, unassigned note and that the out-of-band note payload was encrypted correctly (Section 7.3.3, Voucher Creation).
- **Voucher assignment (purchaser proof):** to attest correct splitting of an unassigned note into an assigned destination note and unassigned change note, with amount conservation and expiration preservation (Section 7.3.3).
- **Voucher redemption (community proof):** to attest authorized spending of an assigned note, nullifier correctness, and correct encrypted liability updates (Section 7.3.3).

Groth16 verification is performed by a dedicated verifier contract, auto-generated from the circuit’s verification key. The voucher contract calls the verifier as part of each state transition (creation, assignment, or redemption). Each circuit (mint, assign, spend) has its own verifier contract. Verification cost is approximately 200K gas per proof, well within L2 transaction budgets.

7.4.2 Token Interfaces and Restricted Transfers

The pool contract V holds reserves of a backing ERC20-style token S and maintains voucher state (trees, nullifiers, liabilities, operator credits). For compatibility, V may expose ERC20-like metadata, but transfers are disabled or tightly restricted to preserve the closed-loop credit model.

7.4.3 Roles and Flows

Voucher Purchase Flow

1. User pays the issuer off-chain (in-app purchase or other channels).
2. Issuer funds the pool via `fundPool`, increasing `deposited` and `available_mint` (Section 7.3.3).
3. User shares an ephemeral note-delivery public key pk_{del} (and a voucher spending address/key material as needed). The issuer encrypts the note payload, sends it over SimpleX (out-of-band), and calls `createVoucher` with a zkSNARK creation proof. V enforces $v \leq available_mint$ and appends the commitment to the current epoch tree.

Operator Redemption and Withdrawals On redemption to operator o , the contract increases `credit[o]` by v_{out} (Section 7.3.3); no stablecoins move.

Later, operator o calls:

`withdrawOperator(o, a).`

The contract must:

1. **check registration/status:** verify o is registered and active in the operator registry;

2. verify $a \leq \text{credit}[o]$;
3. apply the **revenue share policy** specified in the contract (global or per-operator parameters), computing operator share a_o and treasury share $a_T = a - a_o$;
4. decrement $\text{credit}[o] \leftarrow \text{credit}[o] - a$;
5. transfer a_o units of S to the operator payout address and a_T to treasury;
6. update $\text{withdrawn} \leftarrow \text{withdrawn} + a$.

Fee Sponsorship and Relayed Transactions (Paymasters) A privacy pitfall in blockchains is that user-submitted transactions reveal the user’s account address and network-layer metadata at the time of redemption. To avoid leaking linkable identifiers, user redemption transactions should be *relayed*: the user produces a signed redemption payload (and zkSNARK proof) and sends it to an operator/issuer relayer over SimpleX, and the relayer submits the on-chain call.

Centralized top-up fee sponsorship. To make relaying UX-friendly, we propose a *top-up* scheme: on purchase, the issuer withholds a small fee budget from the in-app payment and centrally funds a paymaster service that sponsors gas for voucher assignments and redemptions. Concretely, the issuer funds an ERC-4337 [27] compatible paymaster contract, and users send signed user operations to the relayer which executes them gaslessly via the paymaster.

This preserves privacy (users do not broadcast transactions themselves) while keeping fee management operationally simple (issuer pays the paymaster, not each user).

8 Analysis and Open Questions

The objectives section already records which goals the draft design meets and which it does not; this section gives the direction of the deferred work and the decisions that remain open.

Closing the redemption-privacy gap. Making individual redemptions unobservable, so that only aggregate operator withdrawals are public, has a known solution that the draft does not adopt. Operator payouts become notes: a redemption produces a sealed payout note instead of crediting a named operator, and each operator later withdraws its accumulated total on its own schedule, proving only that the payee is registered. Per-bucket spent totals are written blindly across all live expiration buckets, so a redemption discloses no cohort, and the freshness check moves inside the proof, so the expiration height need not be public. Each step reuses machinery the draft already contains; the cost is a heavier redemption proof, which is why the choice is left open.

Open decisions.

- **Proof system.** The draft assumes 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 measured mobile performance.
- **Expiration-bucket disclosure.** Hiding the bucket entirely requires the blind per-bucket updates above, whose cost grows with the number of live buckets; revealing a coarse bucket (a quarter, say) is far cheaper but narrows a redemption to that purchase window. Bucket size trades anonymity-set size against proof cost.

- **Denomination structure.** Whether to issue notes in a fixed set of denominations — which keeps redemption amounts low-cardinality and so protects unlinkability — and, if so, what that set should be.
- **Note recovery.** Realizing holder recovery from a single seed together with 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.

9 Conclusion

This draft has set out Community Credits as the economic layer 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 light-client queries. The closed-loop shape is deliberate: it lets the system pay operators while keeping users private, and places it within established regulatory exemptions.

The draft meets its economic and solvency objectives, and an assignment hides both the amount and the recipient community. It does not yet make redemptions unobservable as events, and the bucket-aligned expiration that assignment and redemption both publish leaves a coarse purchase-cohort linkage rather than full unlinkability; these gaps are stated, and their remedy sketched rather than adopted. The remaining decisions — proof system, expiration-bucket disclosure, denomination structure, and recovery — are recorded as open. As the design and its parameters settle, this draft will be revised toward a complete specification.

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.

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 Vanguard: 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] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards Privacy in a Smart Contract World. In *Financial Cryptography and Data Security*, 2020.
- [21] FAT SOLUTIONS. Tongo Documentation (Protocol Introduction). <https://docs.tongo.cash/protocol/introduction.html>, accessed 2025.
- [22] E. Ben-Sasson, I. Bentov, Y. Horesh, and S. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. 2018. <https://starkware.co/wp-content/uploads/2022/05/STARK-paper.pdf>, accessed 2025.
- [23] J. Groth. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT*, 2016. <https://eprint.iacr.org/2016/260>

- [24] B. Whitehat, J. Baylina, and M. Bellés. Baby Jubjub Elliptic Curve. EIP-2494 (draft), 2020. <https://eips.ethereum.org/EIPS/eip-2494>
- [25] iden3. rapidsnark: fast Groth16 prover written in C++. <https://github.com/iden3/rapidsnark>
- [26] zkmpo contributors. mopro: Making client-side proving on mobile simple. <https://github.com/zkmpo/mopro>
- [27] V. Buterin, Y. Weiss, K. Payal, D. Kristjansson, and N. Buckner. ERC-4337: Account Abstraction Using Alt Mempool. 2023. <https://eips.ethereum.org/EIPS/eip-4337>