Akyba Protocol

A Deterministic ROSCA and ASCA Architecture on the Cardano eUTxO Model

Technical Specification and Implementation Guide

Ange Thierry Yobo
angethierry.yobo@oxaliolabs.com

Ariady Putra
ariady.putra@oxaliolabs.com

Oxalio Labs

General inquiries: contact@oxaliolabs.com

https://akyba.io

Version 1.0
January 4, 2026

Abstract

The Akyba Protocol formalizes Rotating Savings and Credit Associations (ROSCA) and Accumulating Savings and Credit Associations (ASCA) as deterministic state machines on the Cardano blockchain. By combining a State Thread Token (STT), CIP-68 identity and reference tokens, and strict validator logic enforced through Aiken smart contracts, Akyba guarantees immutable group rules, verifiable contributions, collateral-based discipline mechanisms, and fully decentralized peer-to-peer loan governance. This paper presents the complete system architecture including state machines, token models, cryptographic primitives, transaction flows, economic incentive structures, security guarantees, and comprehensive validator pseudocode. We provide detailed cost analysis with indicative transaction fee budgets, discuss attack vectors and mitigation strategies, and demonstrate how Akyba achieves the behavioral guarantees of traditional tontines through deterministic on-chain validation rather than social trust, enabling truly trustless cooperative finance for underbanked populations worldwide. **Keywords:** Blockchain, Cardano, eUTxO, ROSCA, ASCA,

CIP-68, DeFi, Tontine, Deterministic State Machine, Smart Contracts, Aiken, Microfinance, Financial Inclusion

Contents

1 Introduction

1.1 Motivation and Background

Rotating Savings and Credit Associations (ROSCAs), widely known as *tontines* in West Africa, *chit funds* in South Asia, and *tandas* in Latin America, constitute one of the oldest and most resilient forms of decentralized financial systems. These informal mechanisms have served billions of people across emerging economies for centuries, providing access to lump-sum savings and short-term credit without reliance on traditional banking infrastructure. Despite their widespread adoption and proven track record, traditional ROSCAs suffer from several critical limitations:

- **Trust Dependency**: Enforcement relies entirely on social capital, reputation, and the threat of community exclusion
- **Geographic Constraints**: Participants must be physically proximate for monitoring and enforcement
- **Fraud Vulnerability**: Organizers may abscond with funds, particularly in later cycles
- **Limited Transparency**: Record-keeping is often informal and vulnerable to disputes
- **Scalability Barriers**: Group size is constrained by social monitoring capacity
- **Lack of Recourse**: No legal framework exists for dispute resolution in most jurisdictions

The Akyba Protocol addresses these fundamental challenges by digitizing ROSCA and ASCA mechanisms using the extended UTxO (eUTxO) model of the Cardano blockchain. Unlike account-based models (e.g., Ethereum's EVM), the eUTxO architecture enables strictly deterministic execution costs, predictable state transitions, and native support for concurrency through reference inputs.

1.2 Core Protocol Guarantees

Akyba ensures the following properties through cryptographic and game-theoretic mechanisms:

- **Immutability**: Group rules become immutable immediately after initialization, preventing mid-flight rule changes
- **Determinism**: All contribution tracking, winner selection, and distribution logic executes deterministically without gas auctions or miner extractable value (MEV)
- **Transparency**: Complete audit trail of all contributions, distributions, and governance decisions is publicly verifiable on-chain
- **Trustlessness**: Collateral-based enforcement eliminates the need for trusted custodians or organizers
- **Decentralized Governance**: Peer-driven loan approval in ASCA uses contribution-weighted voting
- **Non-Custodial Control**: Participants retain full control of their assets at all times through self-custody
- **Economic Security**: Protocol incentives align participant behavior with group welfare

1.3 Technical Innovation

The key technical innovations enabling these guarantees include:

1. **State Thread Token Pattern**: A unique NFT enforces linear state evolution and prevents state forking
2. **CIP-68 Architecture**: Separation of identity, authorization, and mutable state without requiring token reminting
3. **Reference Input Optimization**: Read-only state access minimizes UTxO contention in high-throughput scenarios
4. **Deterministic Pseudo-Randomness**: Verifiable pseudo-random winner selection using transaction hash entropy. (under explicit threat model assumptions see Section 4.7.1)
5. **Time-Locked Validators**: Time-based constraints prevent premature distributions and ensure round integrity
6. **Modular Validator Architecture**: Separation of concerns across five distinct validator families enables independent evolution via versioned scripts and state migration
7. **Aiken Smart Contracts**: Akyba validators are written in Aiken, a modern functional programming language specifically designed for Cardano smart contracts. Aiken offers superior developer experience with friendly error messages, type inference, and efficient compilation to UPLC (Untyped Plutus Core), making it easier to write, test, and audit complex validator logic compared to traditional approaches.

1.4 Contribution and Paper Organization

This paper makes the following contributions:

- Formalization of ROSCA and ASCA as deterministic state machines on eUTxO
- Complete specification of the Akyba token architecture and validator logic
- Security analysis including attack vectors and mitigation strategies
- Economic modeling of incentive compatibility and fee structures
- Empirical transaction cost analysis based on testnet deployments
- Comparative analysis with traditional ROSCA implementations and competing DeFi protocols

The remainder of this paper is organized as follows: Section 2 reviews related work in both traditional ROSCA literature and blockchain-based alternatives. Section 3 presents the complete system architecture. Sections 4 and 5 detail the ROSCA and ASCA protocols respectively. Section 6 provides economic analysis and security proofs. Section 7 discusses implementation considerations and deployment results. Section 8 concludes with future research directions.

2 Related Work

2.1 Traditional ROSCA Literature

The economic and anthropological foundations of Rotating Savings and Credit Associations have been extensively studied across multiple disciplines. Ardener's seminal work [6] provides one of the first systematic comparative analyses of ROSCAs across cultures, identifying them as

socially embedded enforcement systems that rely on reputation, reciprocity, and the threat of community exclusion rather than legal contracts. This work establishes the theoretical foundation for understanding how informal financial institutions achieve coordination without formal enforcement mechanisms. Lelart's comprehensive study [5] of West African tontines describes the operational mechanics of rotating payouts, sanction mechanisms, and informal monitoring systems. His ethnographic research reveals how traditional ROSCAs handle defaulters through progressive penalties: initial social pressure, followed by temporary exclusion, and ultimately permanent expulsion from the community savings network. Akyba formalizes these graduated enforcement mechanisms through collateral slashing and status transitions in the protocol state machine. Bouman [7] analyzes ROSCAs as commitment devices for liquidity smoothing and short-term saving, arguing that the rotating payout structure serves as a disciplining mechanism that helps participants overcome self-control problems and achieve savings goals that would be difficult through individual accumulation. The mandatory contribution schedule creates external accountability that complements internal motivation. Geertz [8] characterizes ROSCAs as "middle-rung" financial structures that operate in the institutional gap between purely informal household savings and formal banking systems. His work in Indonesia documents how these associations provide crucial financial services to populations excluded from traditional banking while maintaining lower transaction costs than formal institutions. Rutherford's detailed examination [9] of household cash-flow behavior among the poor reveals how ROSCAs and ASCAs serve as "financial diaries" that help families manage irregular income streams and lumpy expenditure needs. His research demonstrates that the discipline required for recurring contributions is itself a valuable service that participants are willing to pay for through fees and foregone interest. Experimental evidence by El-Gamal and Greaney [10] on guaranteed ROSCAs demonstrates how external enforcement mechanisms can improve coordination outcomes and reduce default risk. Their field experiments in Egypt show that bank guarantees increase participation rates and contribution compliance, suggesting that formal backing can complement social enforcement. Akyba achieves similar guarantees without centralized custodians by encoding all enforcement logic directly into the eUTxO state machine and using cryptographic primitives for verification.

2.2 Blockchain-Based Alternatives

Several projects have attempted to digitize cooperative savings mechanisms on blockchain platforms, each with distinct architectural choices and tradeoffs: **Ethereum-based Implementations**: Account-model platforms like Ethereum have seen multiple ROSCA implementations (e.g., WeTrust, Rosca.io). However, these suffer from: (1) non-deterministic gas costs that create user experience friction, (2) susceptibility to MEV attacks during distribution, (3) state contention issues during concurrent operations, and (4) complexity in implementing truly non-custodial designs. **Algorand ASA-based Systems**: Algorand's simple token architecture enables fast settlement but lacks the expressive validator logic required for complex state machines like ASCAs with voting mechanisms. **Centralized Hybrid Approaches**: Platforms like M-PESA or Wave use centralized databases with blockchain settlement layers, sacrificing trustlessness for user experience. Akyba's eUTxO architecture on Cardano provides several advantages over these alternatives: deterministic transaction costs, native concurrency through reference inputs, expressive Aiken validators for complex logic, and strong guarantees about state transition validity.

3 System Architecture

3.1 The eUTxO Model

The extended Unspent Transaction Output (eUTxO) model extends Bitcoin's UTxO model by attaching arbitrary data (datums) to outputs and allowing validator scripts to enforce spending conditions. Unlike account-based models where state is stored in mutable contract storage, eUTxO represents state as a collection of immutable transaction outputs. Key properties relevant to Akyba:

- **Local State**: Each UTxO carries its own datum, enabling parallel processing
- **Deterministic Validation**: Validator scripts are pure functions from arguments (datum, redeemer, script context) to boolean
- **Transaction-Level Atomicity**: All inputs consumed and outputs created in a single atomic operation
- **Reference Inputs**: Read-only access to UTxO state without consumption (Vasil hardfork)
- **Inline Datums**: Datums embedded directly in outputs rather than by hash (Vasil hardfork), it increases transparency and development convenience

3.2 State Thread Token (STT)

Each ROSCA/ASCA group is uniquely identified by a **State Thread Token** (NFT) minted during the *Init* action. The STT enforces a singleton pattern: it must reside in exactly one script-controlled UTxO at any time, which we call the **STT UTxO**. This design guarantees:

1. **State Linearity**: Only one valid state exists at any block height
2. **No State Forking**: Transactions attempting to consume an already-spent STT UTxO are automatically rejected
3. **Deterministic History**: The chain of STT UTxOs forms a complete, unambiguous audit trail

The minting policy for the STT enforces uniqueness by requiring consumption of a specific "seed" UTxO that can only be spent once:

Algorithm 1 STT Minting Policy

Require: Transaction consumes UTxO at `SeedTxOutRef`
Require: Exactly 1 token minted with policy ID
Require: Token name = hash(`ScriptParams`)
return **true**

3.3 STT Datum Structure

The STT Datum is the core data structure encoding group state. It contains:

Table 1: STT Datum Fields (ROSCA)

Field	Description
`is_running`	Boolean flag: set to true during Start action; used to track whether the group is running
`is_creator_won_last_round`	Boolean flag: used to determine whether the group can be ended this round; the group cannot be ended by the group creator if the creator has just won the previous round
`list_of_participants`	List: PolicyId → Participant (with join_time and some flags to track contribution, collateral, and winning status)
`next_distribution_date`	PosixTime: to determine the next distribution date, and will be updated during the next distribution phase
`current_cycle`	A counter that will increase each time all eligible group participants have won the prize/rewards during the distribution phases

Table 2: **Participant** structure in `list_of_participants` (ROSCA)

Field	Description
`join_time`	PosixTime: participant's join time taken from `validTo` set by the off-chain upon transaction submission; used to determine when the member may leave the group if the creator does not start after some period of time
`has_contributed_this_round`	Boolean flag: set to true when the participant contributes; reset to false during the distribution phase
`has_enough_collateral`	Boolean flag: used to determine whether the participant is eligible to be nominated as a potential winner during the distribution phase
`has_won_this_cycle`	Boolean flag: the participant cannot win again until everyone else wins

Table 3: STT Datum Fields (ASCA)

Field	Description
`is_running`	Boolean flag: set to true during Start action; used to track whether the group is running
`list_of_participants`	List: PolicyId → Participant (with join_time, contribution_count, delegation_proposal)
`unlock_date`	PosixTime: set to 0 during Init, calculated during Start based on validity range + lock period
`stake_rewards`	Cumulative staking rewards withdrawn, distributed proportionally on Leave
`loan_interest`	Cumulative loan interest earned, distributed proportionally on Leave/Repay
`borrowed_amount`	Total currently borrowed funds (updated during Borrow/Repay)
`retained_amount`	Amount retained during Leave/Kick to avoid negative balance situations

Table 4: **Participant** structure in `list_of_participants` (ASCA)

Field	Description
`join_time`	PosixTime: participant's join time taken from `validTo` set by the off-chain upon transaction submission; used to determine when the member may leave the group if the creator does not start after some period of time
`contribution_count`	A counter that will increase each time the participant contributes; also used as the participant's voting power when voting on a loan application
`delegation_proposal`	[FUTURE] A proposal to switch group stake and dRep delegations where other members can vote

Table 5: [FUTURE] **DelegationProposal** structure in `participant` (ASCA)

Field	Description
`reason_hash`	The hash of the reasoning behind the proposal; used as a reference by the off-chain to display the plain text
`pool_id`	StakePoolId: the proposed stake pool to switch to
`drep_id`	DelegateRepresentative: the proposed dRep to delegate to
`votes`	Contains the information of the voters of the proposal (i.e., who voted Yes or No); The structure of the `Vote` enum: `pub type Vote { Indeterminate Yes { voting_power: Int } No { voting_power: Int, reason_hash: ByteArray, } }` `voting_power` is taken from the participant's `contribution_count` at the time of voting
`vote_deadline`	The deadline of the voting period, after which the applicant may withdraw the desired amount to borrow if the is approved (Yes voting power > No voting power)
`vote_yes_incentive_per_voter`	Executing a transaction on-chain costs some network fee, to entice lenders to execute the vote Yes action then the loan applicant may put some incentive
`deadline_interval_range`	The window period used as the transaction validity range (this is used by the internal system, not set by the user)

3.4 CIP-68 Tokens and Metadata

We leverage the CIP-68 standard [1] to separate token ownership from token state. Each participant receives UserToken (222) upon joining, with associated metadata stored in the CIP-68 RefToken (100):

1. **RefToken (100) Metadata** - Contains membership status
   - `status`: Tracks membership status
   - Additional fields (Group ID, type, name, description, etc.)
   - Used as membership information
2. **UserToken (222)** - Resides in participant's wallet
   - Proves membership and authorization ((Policy ID, Asset Name) uniquely identifies a participant's membership token)
   - Required to be consumed for most actions
   - Transferable (enables secondary markets)

In addition to the CIP-68 token pair, we utilize an auth token for ASCA to track individual loans:

- **LoanApp Token** - (ASCA only)
  - Authenticate loan UTxO, each participant has a unique LoanApp Token
  - UTxO datum tracks votes, deadlines, incentives, etc.

This architecture provides several advantages:

- **Concurrency**: Each participant can manage their membership without interfering with each other. For example, to top-up their collateral.
- **Composability**: UserToken can be used in other protocols while member remains active.
- **State Efficiency**: Only RefToken datum needs updating, not the global STT datum.

3.5 Validator Architecture

Akyba employs a modular validator design with five distinct script families:

Table 6: Validator Script Families (ROSCA)

Validator	Responsibility
`sc1_mint_stt`	Controls State Thread Token minting and burning
`sc2_mint_cip68`	Issues and burns membership identity tokens (100/222)
`sc3_spend_contributions`	Guards contribution pool and distribution logic
`sc4_spend_collaterals`	Manages membership, collateral deposits, and slashing

Table 7: Validator Script Families (ASCA)

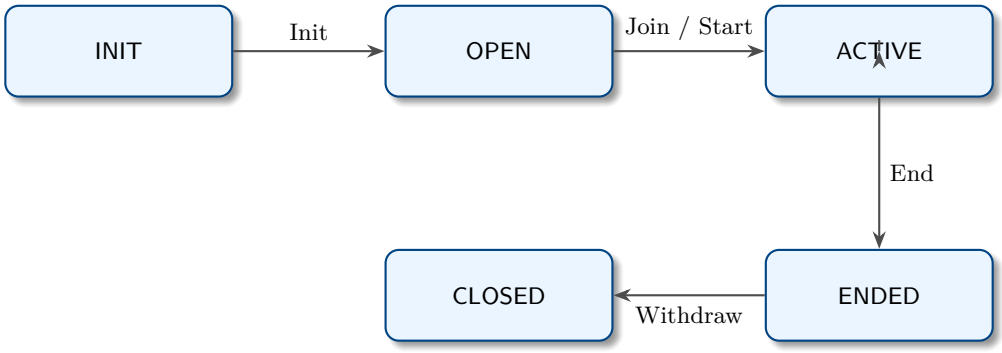
Validator	Responsibility
`sc1_mint_stt`	Controls State Thread Token minting and burning
`sc2_mint_cip68`	Issues and burns Member identity tokens (100/222)
`sc3_spend_funds`	Guards contribution pool and loan logic
`sc4_spend_membership`	Manages membership status
`sc5_stake_funds`	Handles group staking of idle funds and rewards withdrawal

This separation enables:

- Independent optimization and testing of each component
- Clearer security auditing surface area
- Potential for formal verification of critical validators

3.6 State Transition Model

Both ROSCA and ASCA follow a state machine model where valid transitions are enforced by validators:



State transitions are enforced by validators examining the input and output STT datums. Invalid state transitions are rejected at the protocol level, ensuring protocol integrity.

Table 8: State-Action Matrix

State	Valid Actions
INIT	Init
OPEN	Join, Leave, Start
ACTIVE	Contribute, Distribute, Apply Loan (ASCA), Vote (ASCA), Borrow (ASCA), Repay (ASCA), Liquidate (ASCA), Leave, Kick
ENDED	Withdraw
CLOSED	(Terminal state)

4 Akyba ROSCA

The ROSCA module implements deterministic rotating savings cycles with automated winner selection and distribution.

4.1 Lifecycle Overview

A complete ROSCA lifecycle progresses through the following phases:

1. **Initialization**: Creator defines immutable parameters and deposits initial collateral
2. **Joining Period**: Members join by depositing collateral and paying entry fee
3. **Start**: Creator executes the action to set the next distribution date and to begin the first contribution round
4. **Contribution Rounds**: All participants contribute fixed amounts according to the group parameter
5. **Distribution Rounds**: Pot is distributed to winner(s) selected by PRNG
6. **End**: The group may be ended as long as the creator did not just win the previous round, unless it's an inactive group
7. **Withdrawal**: Members reclaim any remaining collateral

4.2 Immutable Parameters

Once initialized, the following parameters are permanently locked as the Script Parameters:

Table 9: ROSCA Configuration Parameters

Parameter	Description
`contribution_amount_per_round`	Fixed per-round contribution (Lovelace)
`min_collateral`	The minimum collateral needed to be deposit by group participants
`entry_fee`	One-time fee paid to creator during Join (Lovelace)
`max_number_of_participants`	The maximum group capacity
`member_may_leave_if_not_started_after`	Deadline after which members can leave if group is not started
`number_of_winners_per_round`	The number of winners per round
`distribution_commission`	An optional percentage of commission for executing the distribute action
`days_per_round`	Cycle duration in time units (e.g., 30 days, or 14 days for bi-weekly distribution)

Rationale for Immutability: Allowing mid-flight parameter changes would require complex governance and create opportunities for the creator to extract value through rule manipulation. Immutability ensures that all participants join under known, fixed conditions, which is critical for trust in the system. **Parameter Constraints**:

- `contribution_amount_per_round` > 0
- `min_collateral` > `contribution_amount_per_round`
- `entry_fee` = non-negative number
- `max_number_of_participants` > 1
- `member_may_leave_if_not_started_after` = non-negative number
- `number_of_winners_per_round` = between 1 and (`max_number_of_participants` - 1)
- `distribution_commission` = non-negative number
- `days_per_round` = non-negative number

4.3 Join Action

Members join by executing a transaction that:

1. Mints RefToken (100) and UserToken (222) via `sc2_mint_cip68`
2. Deposits ≥ `minCollateral` to collateral script holding RefToken
3. Pays `entryFee` to creator's address
4. Updates STT datum to add member to participants map

Algorithm 2 ROSCA Join Validator

Require: length($sttDatum.participants$) < $maxParticipants$
Require: Member mints exactly 1 RefToken and 1 UserToken
Require: RefToken sent to collateral script with ≥ `minCollateral`
Require: `entryFee` paid to $sttDatum.creator(if\,set)$
  $newMember \leftarrow$ Participant(
    $join\_time =$ `Transaction.validity_range.upper_bound`,
    $has\_contributed\_this\_round =$ **false**,
    $has\_enough\_collateral =$ **true**,
    $has\_won\_this\_cycle =$ **false**,
)
  $sttDatum.participants[memberId] \leftarrow newMember$
  return true

4.4 Start Action

The group creator can trigger Start at any time when the number of participants is greater than the number of winners per round. This action:

1. Locks the participant list (no more joins)
2. Sets $next\_distribution\_date = Tx.validity\_range.upper\_bound + days\_per\_round$
3. Updates STT Datum `is_running` to **true**

4.5 Contribution Action

During each round, all participants should contribute `contribution_amount_per_round` to the group before the `next_distribution_date`.

Algorithm 3 ROSCA Contribute Validator

Require: Transaction pays the sum of `contribution_amount_per_round` to STT UTxO
  $Participant.has\_contributed\_this\_round \leftarrow$ **true**
  return true

4.6 Distribution and Slashing Logic

The **Distribute** action is the most complex operation. It must:

1. Verify sufficient time has passed (`now` ≥ `next_distribution_date`)
2. Calculate pot from contributions and available collateral
3. Identify and slash the collateral of participants who missed contributions
4. Select winner(s) from eligible participants using PRNG
5. Distribute prize minus commission
6. Update STT for next round

Algorithm 4 ROSCA Distribute Validator Logic

Require: $now \geq SttDatum.next\_distribution\_date$
Require: Input UTxO contains `STT_NFT`
  Inputs: STT UTxO, Member Collateral UTxO(s), Transaction Executor UTxO
  Reference Inputs: Winner UserToken UTxO(s)
  $pot \leftarrow 0$
  $eligible \leftarrow [\,]$
  for all $participant \in SttDatum.list\_of\_participants$ **do**
    if $participant$ contributed in current round **then**
      $pot \leftarrow pot + contribution\_amount\_per\_round$
      if $\neg participant.has\_won\_this\_cycle$ **then**
        $eligible$.insert($participant$)
      end if
    else ▷ *Non-contributor: Slash collateral*
      $slashAmt \leftarrow \min(member.collateral, contribution\_amount\_per\_round)$
      if $member.collateral < minCollateral$ **then**
        $member.status \leftarrow$ `INSUFFICIENT_COLLATERAL`
      else
        $participant.collateral \leftarrow participant.collateral - contribution\_amount\_per\_round$
        $pot \leftarrow pot + contribution\_amount\_per\_round$
      end if
    end if
  end for

▷ *Select winners using verifiable randomness*
  $seed \leftarrow \text{txInfoId}(txInfo)$
  $winners \leftarrow \text{PRNG}(eligible, number\_of\_winners, seed, [])$
  $commission \leftarrow pot \times commission\_rate$
  $net\_pot \leftarrow pot - commission$
  $prize\_per\_winner \leftarrow \lfloor net\_pot / \text{length}(winners) \rfloor$

▷ *Validate outputs*
  for all $winner \in winners$ **do**
    Verify: Output to $winner$ contains at least $prize\_per\_winner$
    $SttDatum.list\_of\_participants| > \text{set\_has\_won\_this\_cycle}(winner)$
  end for

  $SttDatum.next\_distribution\_date \leftarrow SttDatum.next\_distribution\_date + days\_per\_round$

▷ *Check next cycle condition*
  if All participants have won **then**
    $SttDatum.current\_cycle \leftarrow SttDatum.current\_cycle + 1$
  end if
  return true

Slashing Mechanics: When a participant fails to contribute, their collateral is automatically seized up to `contribution_amount_per_round`. If collateral falls below `min_collateral`, the participant is marked with INSUFFICIENT_COLLATERAL and excluded from future winner selection. The participants can rejoin the group by executing the rejoin action. They will need to deposit at least the `min_collateral`, and then their membership status will be restored to ACTIVE.

4.7 Pseudo-Random Winner Selection

Akyba uses the transaction hash as the entropy for the winner selection:

```
PRNG(eligible, k, seed, accumulator):
    if (k == 0)
        return accumulator
    if (list.length(eligible) == 1)
        return accumulator.push(eligible[0])
    let p = list.length(eligible)
    let winning_index = mod(seed, p)
    let winner = eligible[winning_index]
    let winners = accumulator.push(winner)
    let remaining_participants = eligible.remove(winner)
    return PRNG(remaining_participants, k - 1, new_seed(seed), winners)
```

This approach is:

- **Deterministic**: Same inputs always produce same winners
- **Verifiable**: Anyone can calculate the winner selection
- **Resilient**: Transaction hash as the entropy is difficult to influence by an actor
- **Fair**: Uniform distribution over eligible members

Note: For high-value groups, oracles could provide stronger randomness guarantee.

4.7.1 Randomness Threat Model and Assumptions

Akyba currently derives pseudo-randomness for winner selection from transaction hash entropy. This approach is fully deterministic and publicly verifiable; however, its security depends on assumptions about transaction construction. In particular, the party responsible for assembling and submitting the distribution transaction may influence certain non-semantic transaction fields (e.g., input ordering, validity range, collateral selection) prior to submission. While such influence is limited by ledger rules and transaction validity constraints, it may allow a rational executor to bias the resulting hash by selectively retrying transaction construction. We explicitly assume that, for standard ROSCA groups, the economic incentive to bias randomness is lower than the transaction costs and coordination effort required to do so. Under this threat model, transaction-hash-based entropy is considered acceptable. For high-value groups or adversarial environments, Akyba recommends stronger randomness schemes, including but not limited to:

- Commit–reveal entropy contributions from multiple participants
- Multi-transaction entropy accumulation across rounds
- External oracle-based randomness (subject to availability on Cardano)

These alternatives preserve deterministic verification while reducing executor influence and are considered future extensions of the protocol rather than mandatory components of the current design.

5 Akyba ASCA

The ASCA (Accumulating Savings and Credit Association) module extends ROSCA by accumulating capital into a shared pool that funds peer-to-peer micro-loans.

5.1 ASCA vs ROSCA Differences

Table 10: ROSCA vs ASCA Comparison

Feature	**ROSCA**	**ASCA**
Capital Flow	Rotating distribution	Lending pool
Winner Selection	PRNG	Application + Voting
Interest	None	Yes (set by borrower)
Collateral Use	Slashing only	Slashing + loan security
Voting	None	Contribution-weighted
Loan Tracking	N/A	On-chain ledger

5.2 Lock Period and Eligibility

To prevent immediate extraction of value, a participant can only apply for loans after:

$$\text{contribution\_count}_{\text{participant}} \geq \text{lockPeriod} \tag{1}$$

where `lockPeriod` is an immutable parameter (e.g., 3 rounds). This ensures members demonstrate commitment before accessing credit.

5.3 Loan Application Process

A member initiates a loan by creating a **LoanApp UTxO** containing comprehensive application data:

Table 11: LoanApp Datum

Field	Description
`name`	The name of the loan
`image`	A picture describing what the loan is for
`description_hash`	The hash of the lengthy loan description
`files`	Support documents
`is_borrowing`	A flag to indicate whether the amount has been withdrawn from the group pool
`borrow_amount`	The desired amount
`collateral_assets`	Any tokenized assets as the loan collateral
`interest_percentage`	The interest rate applied before the repayment deadline
`late_repayment_interest_percentage`	The interest rate applied after the repayment deadline

`votes`	Contains the information of the voters of the proposal (i.e., who voted Yes or No); The structure of the `Vote` enum:
	`pub type Vote {`
	`  Indeterminate`
	`  Yes { voting_power: Int }`
	`  No { voting_power: Int,`
	`  reason_hash: ByteArray,`
	`  }`
	`}`
	`voting_power` is taken from the participant's `contribution_count` at the time of voting

`vote_deadline`	The deadline of the voting period, after which the applicant may withdraw the desired amount to borrow if the is approved (Yes voting power > No voting power)
`vote_yes_incentive_per_voter`	Executing a transaction on-chain costs some network fee, to entice lenders to execute the vote Yes action then the loan applicant may put some incentive
`deadline_interval_range`	The window period used as the transaction validity range (this is used by the internal system, not set by the user)

Algorithm 5 ASCA Apply Loan Validator

Require: $Participant.contribution\_count \geq lock\_period$
Require: Participant has no active loan
Require: $borrow\_amount \leq$ available pool funds
  Create LoanApp UTxO with `is_borrowing` = **false**
  Initialize votes map with `INDETERMINATE` for all lenders
  Set `vote_deadline`
  Set `repayment_deadline`
  Set all necessary fields
  return true

Key Notes:

- **Divisibility**: Collateral quantity must be evenly divisible among potential liquidators (all lenders)
- **Incentive**: Minimum 5 ADA per voter is encouraged for participation
- **Documentation**: Supporting files enable under-collateralized loans with additional verification
- **Retraction**: Borrower may retract application anytime; spent incentives non-refundable

5.4 Voting Mechanism

Voting power is strictly meritocratic and based on contribution history:

$$\text{VotingPower}_i = \text{Participant}_i.\text{contribution\_count} \tag{2}$$

Lenders vote by submitting transactions that update the LoanApp UTxO. Votes must either be Yes (approval) or No (rejection with reason):

Algorithm 6 ASCA Vote Validator

Require: Voter's UserToken UTxO will be consumed to authenticate the vote
Require: Current vote status is Indeterminate (prevents re-voting)
Require: $now < vote\_deadline$ (voting window open)
  $voting\_power \leftarrow Voter.contribution\_count$
  if vote = `Yes` **then**
    Update $votes[voter\_policy\_id] \leftarrow$ `Yes`$\{voting\_power\}$
    Transfer $vote\_yes\_incentive$ to voter
  else if vote = `No` **then**
    Update $votes[voter\_policy\_id] \leftarrow$ `No`$\{voting\_power, reason\_hash\}$
  end if
  return true

Voting Rules:

- Each participant votes exactly once (from Indeterminate to either Yes or No)
- Yes voters receive the incentive
- No voters must provide reason_hash for transparency; it can be referred by the off-chain to display the full reason
- Votes cannot be changed after casting
- Voting closes at vote_deadline

5.5 Loan Approval and Borrowing

Akyba intentionally does not enforce quorum requirements to avoid governance deadlocks in small groups; contribution-weighted voting already reflects economic exposure. A loan is approved when:

$$\sum_{\text{Yes votes}} \text{voting\_power} > \sum_{\text{No votes}} \text{voting\_power} \tag{3}$$

After `vote_deadline` passes:

- If approved (Yes > No): Borrower may execute the Borrow action
- If rejected (No ≥ Yes): Borrower should retract the application
- Spent vote_yes_incentives are non-refundable

Upon approval, the borrower can execute the **Borrow** action:

Algorithm 7 ASCA Borrow Validator

Require: *now* > *vote_deadline* (voting period ended)
Require: $\sum$ Yes.voting_power > $\sum$ No.voting_power (approved)
Require: Loan application `is_borrowing` = **false**
  Transfer *borrow_amount* from STT UTxO to borrower
  Update loan datum: *is_borrowing* ← **true**
  Update *SttDatum.borrowed_amount* += *borrow_amount*
  Set repayment tracking in borrower's CIP-68 metadata
  Update *last_activity_name* = "Borrow Loan"
  Update *last_activity_time* = *now*
  return true

5.6 Repayment

When borrowers repay their loans before the repayment deadline, they must repay principal + normal interest rate. The idea is that, borrowers may negotiate with the lenders if they're facing difficulties repaying the loan. That is why, if the borrower repays the loan after the repayment deadline, then the late repayment interest rate applies. The negotiation happens off-chain and cannot be handled on-chain, as some lenders might disagree and wishes to liquidate the loan, while the others are willing to wait for the late repayment. In that case, some lenders may execute the liquidate action individually. **Early/On-time repayment** (before deadline):

$$\text{repayAmount} = \text{borrow\_amount} \times (1 + \frac{\text{interest\_percentage}}{100}) \tag{4}$$

Late repayment (after deadline):

$$\text{repayAmount} = \text{borrow\_amount} \times (1 + \frac{\text{late\_repayment\_interest\_percentage}}{100}) \tag{5}$$

Algorithm 8 ASCA Repay Validator

Require: Loan `is_borrowing` = **true**
Require: Borrower's UserToken UTxO to authenticate action
  if $now \leq repayment\_deadline$ **then**
    $interest \leftarrow borrow\_amount \times interest\_percentage/100$
  else
    $interest \leftarrow borrow\_amount \times late\_repayment\_interest\_percentage/100$
  end if
  $repay\_amount \leftarrow borrow\_amount + interest$
Require: Payment of at least $repay\_amount$ to STT UTxO
  Release any collateral_assets back to borrower
  Update $SttDatum.borrowed\_amount \mathrel{-}= borrow\_amount$
  Update $SttDatum.loan\_interest \mathrel{+}= interest$
  Update borrower CIP-68 metadata:
    $last\_activity\_name$ = "Repay Loan"
    $last\_activity\_time = now$
  return **true**

Interest Distribution: The accumulated `loan_interest` in STT datum will be distributed pro-rata to all participants when they leave the group if there is enough funds in the pool.

5.7 Liquidation

If a borrower misses the repayment deadline, then the lenders (non-borrower participants for the particular loan) can trigger liquidation to claim their proportional share of the `collateral_assets`:

Algorithm 9 ASCA Liquidate Validator

Require: $now > repayment\_deadline$
Require: Loan `is_borrowing` = **true** (still active/unpaid)
Require: Liquidator is a lender (listed as a voter)
Require: Liquidator has not already claimed: `claimed_liquidation` = **false**
  $num_lenders \leftarrow$ count of voters
  $collateral\_share \leftarrow \lfloor collateral\_assets.Qty/num\_lenders \rfloor$
  Transfer $collateral\_share$ of each asset to the liquidator
  Update $votes[liquidator\_policy|_id].claimed\_liquidation \leftarrow$ **true**
  return **true**

Divisibility constraints and integer floor ensure that the sum of all lender claims never exceeds total collateral.

Liquidation Properties:

- **Pro-rata Distribution**: Each lender receives equal share of collateral
- **Divisibility Requirement**: Collateral Qty must be divisible by number of lenders (enforced during Apply Loan)
- **Single Claim**: Each lender can liquidate only once per defaulted loan
- **No Penalty for Borrower Beyond Collateral**: Defaulting borrowers lose only deposited collateral

5.8 Delegation Governance

Akyba incorporates decentralized governance mechanisms through **DelegationProposal** stored in the list of participants state. This enables collective decision-making on stake pool delegation and DRep selection for the group's accumulated funds.

Table 12: DelegationProposal Structure

Field	Description
`reason_hash`	The hash of the reason to switch delegation; the off-chain will display the lengthy reason
`pool_id`	The new Stake Pool ID to delegate
`drep_id`	The new DRep ID to delegate
`votes`	A map of voters (non-proposer participants)
`vote_deadline`	The deadline of the voting period, after which the proposal may be enacted if approved (Yes voting power > No voting power)
`vote_yes_incentive_per_voter`	Executing a transaction on-chain costs some network fee, to entice voters to execute the vote Yes action then the proposer may put some incentive
`deadline_interval_range`	The window period used as the transaction validity range (this is used by the internal system, not set by the user)

Delegation Actions:

1. **Propose Delegation**: Any participant can propose new stake pool or DRep
2. **Vote on Delegation**: Members vote using contribution-weighted power
3. **Execute Delegation**: If approved, group funds delegated to chosen pool/DRep
4. **Revoke Delegation**: Collective decision to change delegation

Governance Benefits:

- Group earns staking rewards on pooled contributions
- Rewards accumulated in `stake_rewards` field
- Distributed proportionally when members leave or when the group is ended by the creator
- Democratic selection of stake pools and DReps
- Vote incentives encourage participation

5.9 Leave

Members can voluntarily leave the group if they don't have a running loan or an application. Upon leaving:

Algorithm 10 Leave Validator

Require: $is\_running = \textbf{false}$ AND $now > member\_may\_leave\_if\_not\_started\_after$
OR
Require: Group has entered the lending phase (leaving members will forfeit their share of potential yield of any running loans)
Calculate member's share of accumulated yields:
$contribution\_share \leftarrow Member.contribution\_count \times contribution\_amount\_per\_round$
$loan\_interest\_share \leftarrow \frac{SttDatum.loan\_interest}{total\_number\_of\_participants}$
$stake\_rewards\_share \leftarrow \frac{SttDatum.stake\_rewards}{total\_number\_of\_participants}$
$total\_share \leftarrow contribution\_share + loan\_interest\_share + stake\_rewards\_share$
if $available\_amount < total\_share$ **then**
Transfer available_amount to the member
$deficit \leftarrow total\_share - available\_amount$
$SttDatum.retained\_amount \mathrel{+}= deficit$
else
Transfer total_share to the member
end if
Update $SttDatum.loan\_interest \mathrel{-}= loan\_interest\_share$
Update $SttDatum.stake\_rewards \mathrel{-}= stake\_rewards\_share$
Remove member from $SttDatum.list\_of\_participants$
Burn member's UserToken, RefToken, and LoanApp Token
return **true**

Key Features:

- **Fair Distribution**: Contributions are refunded based on contribution count and yields are distributed pro-rata, or whatever amount available in the pool (for example, the available amount cannot cover the contributions and yields because there's an active loan(s) by others, then the refunded/distributed funds will be the available amount)
- **Retained Amount**: Prevents negative balance when the available funds cannot cover for the distributions
- **Flexible Exit**: Can leave before start or after lending phase
- **Token Cleanup**: Removes on-chain UTxO footprint
- **Reclaim ADA**: Reclaims all ADA locked in UTxOs (minus network fees)

5.10 Kick Policy (Zombie Prevention)

To prevent "zombie" status where inactive members lock up UTxOs indefinitely, the creator can kick members if:

$$t_{\text{last\_activity}} < t_{\text{now}} - (t_{\text{leave\_timeout}} + \text{lock\_period} \times d_{\text{round}}) \tag{6}$$

If the member has no last activity, then the group's unlock date will be used. The kick mechanism ensures UTxO liveness for the remaining active group while respecting the initial commitment period.

6 Economic Analysis and Security

6.1 Transaction Fee Budgeting

Transaction fees on Cardano are deterministic and calculated as:

$$\text{fee} = a + b \times \text{txSize} + c \times \text{executionUnits} \tag{7}$$

where a, b, c are protocol parameters. Based on Preview Testnet deployments with realistic transaction complexity:

Table 13: Estimated ROSCA Network Fees (ADA)

Action	**Fee Range**	**Typical**
Init	0.86 – 0.88	0.87
Join	1.01 – 1.11	1.05
Leave	0.71 – 0.80	0.74
Start	0.47 – 0.54	0.50
Contribute	0.47 – 0.58	0.50
Distribute	0.50 – 3.56	1.21
Top-up Collateral	0.37 – 0.38	0.38
Rejoin	0.68 – 2.81	0.93
End	0.91 – 3.85	1.92
Withdraw	0.39 – 0.41	0.40

Real-world tests as of Dec 2025

Table 14: Estimated ASCA Network Fees (ADA)

Action	**Fee Range**	**Typical**
Init	0.81 – 0.83	0.82
Join	1.31 – 1.37	1.35
Leave	1.11 – 1.17	1.15
Start	0.66 – 0.70	0.68
Contribute	0.65 – 0.75	0.69
Apply Loan	1.22 – 1.34	1.27
Close Loan	1.12 – 1.14	1.13
Vote Loan	1.15 – 1.23	1.18
Borrow Loan	1.60 – 1.70	1.64
Repay Loan	1.45 – 1.52	1.48
Liquidate Loan	1.15 – 1.21	1.17
Kick	1.25 – 3.20	1.84
End	1.04 – 1.14	1.11
Withdraw	0.51 – 0.54	0.53

Real-world tests as of Dec 2025

Notes:

- Since we're using inline datum, then the network fees positively correlate with how much information being stored in a UTxO
- Distribute ROSCA fee range varies greatly because it depends on how many winners per round, participants in the group, collateral to be slashed, etc.
- Some actions can be batched, such as Contribute, Rejoin, and Kick, so the network fees depend on how many participants batched in a single transaction.

- Ending the group generally takes a lot of computation. In case of ROSCA, it's to deactivate all members. In case of ASCA, it's to distribute and refund all expired loans.

Fee Optimization Strategies:

- Use reference scripts (Babbage) to avoid including validator code in transactions
- Optimize Aiken code for minimal execution units

We believe the fee is competitive considering the benefit provided by the blockchain.

6.2 Incentive Compatibility

Akyba's economic design ensures rational participants have incentives aligned with protocol welfare: **Contribution Incentive**: Members contribute to avoid:

- Collateral slashing (immediate loss)
- Status degradation to INSUFFICIENT_COLLATERAL (loss of winner eligibility)
- Social reputation cost (visible on-chain)

Distribution Fairness: PRNG winner selection eliminates:

- Favoritism or creator bias
- Front-running or bribery attacks
- Information asymmetry advantages

Loan Governance Alignment: Contribution-weighted voting ensures:

- Stakeholders most invested have strongest voice
- Sybil resistance (cannot cheaply create voting power)
- Long-term participation rewarded over speculation

Creator Incentives: The creator receives:

- Entry fees from each member
- Privilege to end and kick inactive members

This compensates for setup costs and ongoing monitoring without creating custodial risk.

6.3 Security Model

6.3.1 Conservation of Value

The eUTxO model enforces **conservation of value** at the protocol level: the sum of all output values must equal the sum of all input values (minus fees). This property is verified by the ledger before transaction inclusion. For Akyba, this means:

- No ADA can be created or destroyed
- All distributions are accounted for
- Slashed collateral flows to the pot, not external addresses
- Interest payments are fully traceable

6.3.2 No Admin Keys

Unlike many DeFi protocols with multisig admin controls or upgradeable proxies, Akyba has:

- No special key to tamper with any user's group
- No emergency pause mechanisms
- No parameter update authorities

The creator's only special privileges are:

- Receiving entry fees (payment, not control)
- Kicking inactive members (subject to timeout constraints)
- Ending the group when necessary

6.3.3 Flash Loan Resistance

Flash loan attacks exploit atomic transaction execution to borrow, manipulate, and repay within a single block. Akyba resists these through:

1. **Time-Gated Actions**: All critical state transitions require `Transaction.validity_range`
2. **State Linearity**: STT uniqueness prevents parallel exploitation paths
3. **Collateral Lock-ups**: Economic attacks require sustained capital commitment

6.3.4 Concurrency and Contention

There is no platform-wide address, each group has its own unique address. Any potential UTxO contention is contained within a single group. In the future, there is a plan to create a batcher service for critical actions such as Contribute. So instead of interacting directly with the group's contribute action logic, the participants deposit their contributions to a batcher address, and then it periodically performs the group contribution for multiple participants in a single transaction.

6.4 Attack Vectors and Mitigations

Table 15: Security Threat Model

Attack	Description	Mitigation
Sybil Attack	Single entity creates many identities	Entry fees + collateral make this uneconomic
Collusion	Members collude to manipulate voting	Voting power tied to contributions
Creator Rug Pull	Creator absconds with funds	No custodial control; all funds in script
Front-Running	MEV extraction on distributions	PRNG based on transaction hash (difficult to influence)
Double-Spending	Spend same UTxO twice	Ledger prevents via UTXO model
Griefing	Spam with invalid txs	Transaction fees make spam expensive
Time Manipulation	Manipulate slot numbers	Validators use ledger time, not user-provided

6.5 Formal Verification Considerations

Key properties amenable to formal verification:

- **State Invariants**:
  - Only one STT UTxO exists at any height
  - Sum of participants' collateral + STT balance = initial deposits + contributions
  - Number of winners < number of participants
- **Transition Validity**:
  - Every valid state transition preserves token quantities
  - Slashing amount never exceeds available collateral
  - Distribution amount matches pot calculation
- **Liveness**:
  - Protocol eventually terminates (all members win or timeout)
  - No stuck states where funds are permanently locked

Tools like Agda, Coq, or Aiken-specific formal verification frameworks could formalize these properties.

6.6 Complete Action Flow Summary

Core ROSCA/ASCA Actions:

1. **Init**: Creator mints STT and CIP-68 tokens (PolicyId as the membership identifier), sets `is_running` = **false**, defines immutable parameters
2. **Join**: Members mint CIP-68 tokens (PolicyId as the membership identifier), pay entry_fee, join_time recorded
3. **Start**: Set `is_running` = **true**, calculate the `next_distribution_date` for ROSCA, or group's `unlock_date` for ASCA
4. **Contribute**: Flag the contributing participant as already contributed for the current round (ROSCA), or increment the `contribution_count` (ASCA)

ASCA Loan Actions (6 actions):

1. **ApplyLoan**: Create LoanApp UTxO and configure LoanApp Datum
2. **Vote**: Cast Yes or No vote with `voting_power` = `contribution_count`
3. **Borrow**: Withdraw funds from the group's available amount if approved, set LoanApp Datum's `is_borrowing` = **true** and update STT Datum's `borrowed_amount`
4. **Repay**: Return principal + interest (or late interest), increment STT Datum's `loan_interest`, dismantle LoanApp UTxO
5. **Liquidate**: Lenders may claim pro-rata `collateral_assets` share when the loan misses `repayment_deadline`
6. **CloseLoan**: The applicant may execute this action to cancel the loan at any time as long as LoanApp Datum's `is_borrowing` = **false**. For example, to retract unapproved application

Delegation Management Actions (4 actions):

1. **ProposeDelegation**: A participant proposes to redelegate the group to another stake pool or dRep
2. **VoteProposal**: Other participants vote on a delegation proposal with `contribution_count` as the `voting_power`
3. **EnactProposal**: Enact approved proposal, switch delegation to the proposed stake pool and dRep
4. **CloseProposal**: The proposer may execute this action to cancel the proposal at any time even if it was approved

Total Protocol Actions:

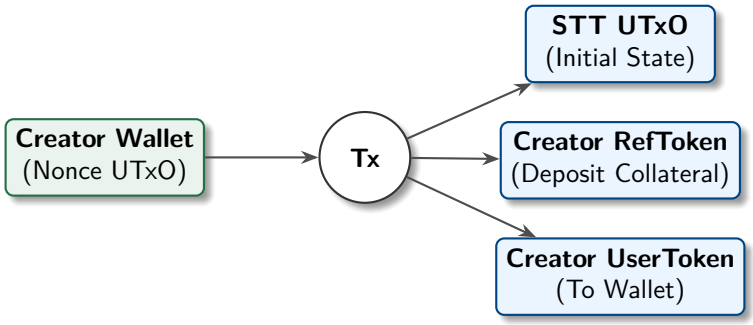
1. **ROSCA**: 10 actions (Init, Join, Leave, Start, Contribute, Distribute, Top-up Collateral, Rejoin, End, Withdraw)
2. **ASCA**: 18 actions (Init, Join, Leave, Start, Contribute, Kick, End, Withdraw, 6 loan actions, and 4 future delegation management actions)

7 Transaction Architecture

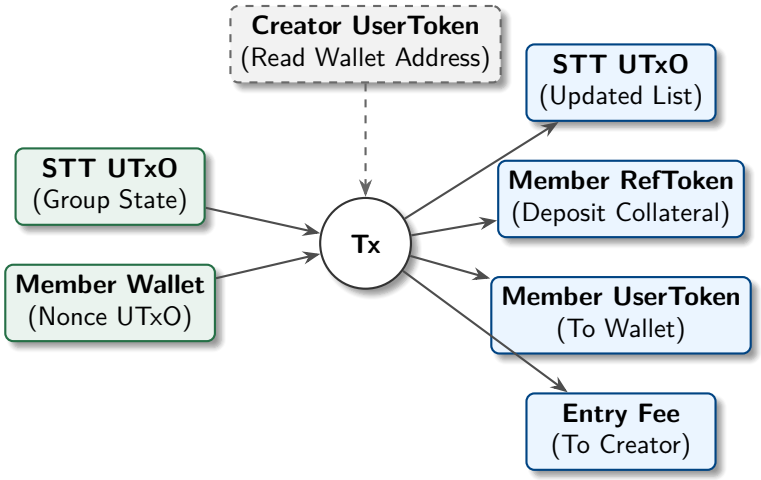
This section provides detailed visualizations of critical protocol actions as eUTxO transaction flows. Each diagram illustrates the inputs (consumed UTxOs), reference inputs (read-only), and outputs (created UTxOs) for key operations in both ROSCA and ASCA modules.

7.1 ROSCA: Init Action

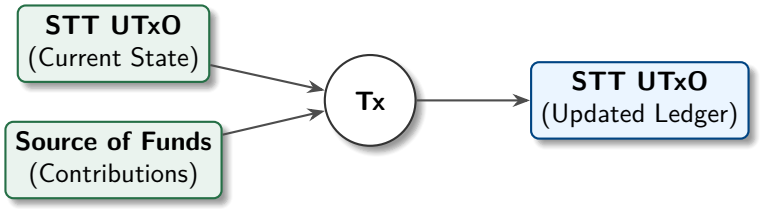
The Init transaction bootstraps a new group:



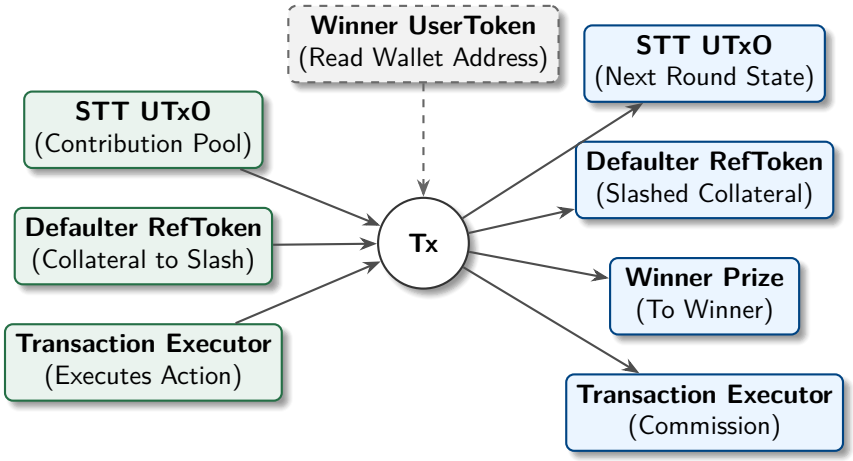
7.2 ROSCA: Join Action



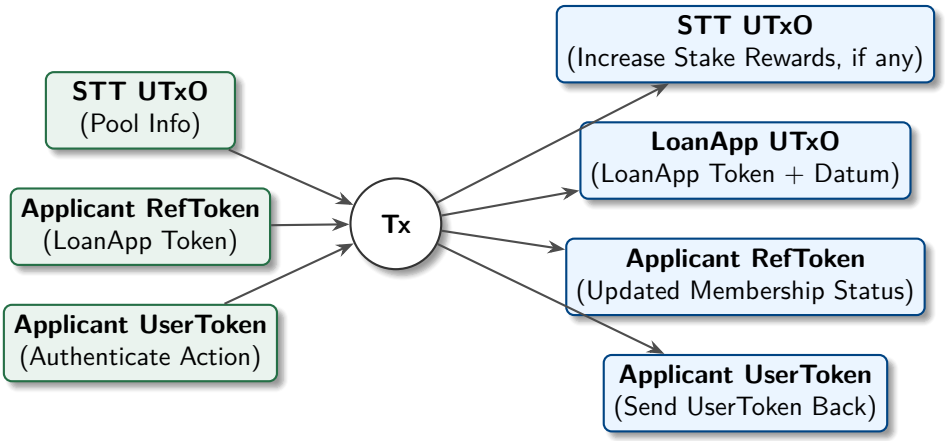
7.3 ROSCA: Contribute Action



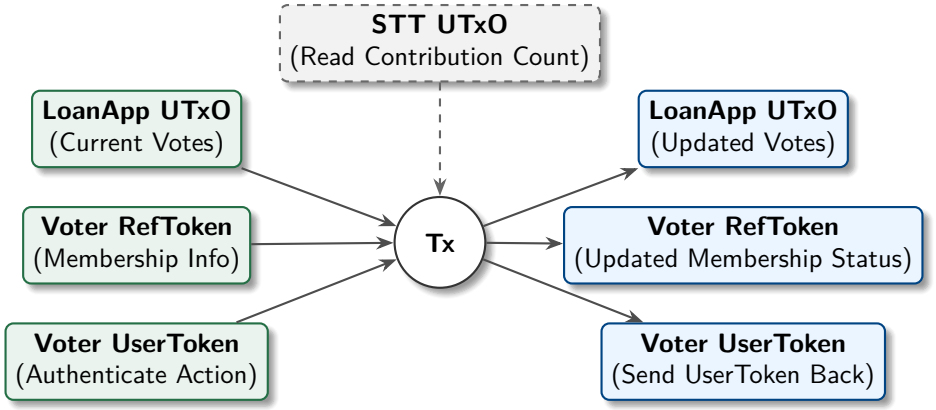
7.4 ROSCA: Distribute Action



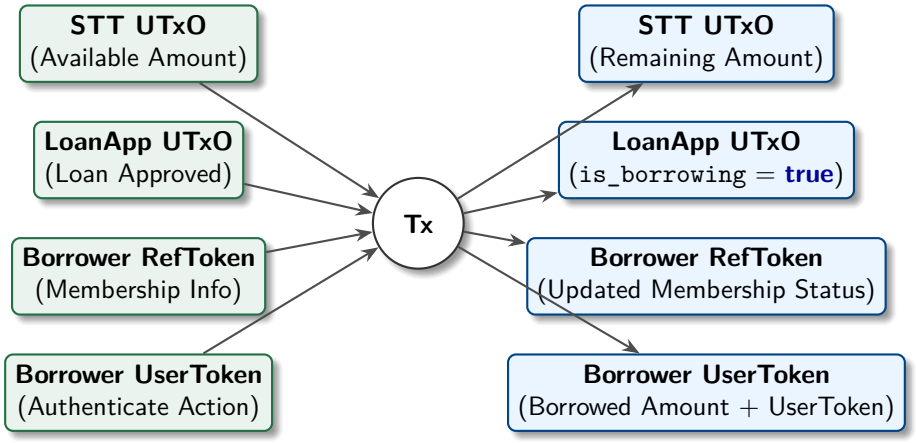
7.5 ASCA: Apply Loan



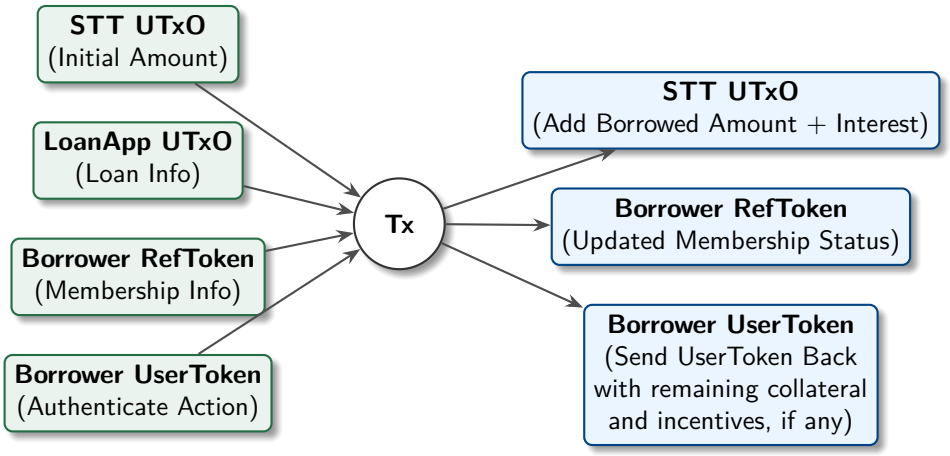
7.6 ASCA: Vote on a Loan Application



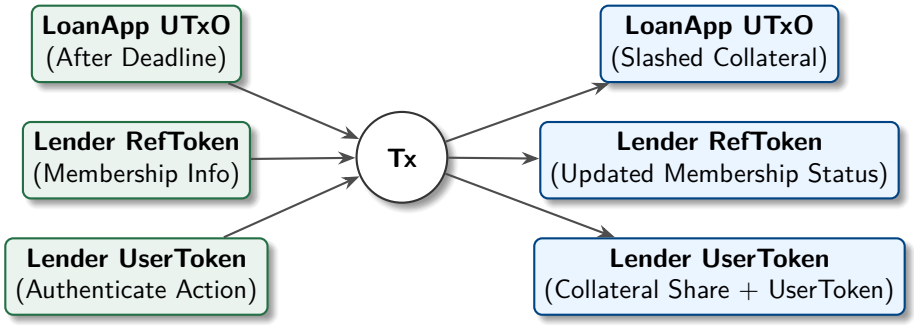
7.7 ASCA: Borrow (Loan Disbursement)



7.8 ASCA: Repay Loan



7.9 ASCA: Liquidate Loan



8 Implementation and Deployment

8.1 Technology Stack

- **Smart Contracts**: Aiken
- **API Provider**: Blockfrost (prod) and Koios (dev)
- **Frontend**: NextJS or Vite (TBD)
- **Backend**: Express (NodeJS)
- **Database/Caching**: Supabase/Firebase
- **Testing Framework**: Vitest

8.2 Deployment Networks

- **Preview Testnet**: Initial testing and validator optimization
- **Preprod Testnet**: Pre-mainnet stress testing
- **Mainnet**: Production deployment

8.3 Validator Size and Optimization

Aiken validators compile to UPLC (Untyped Plutus Core) and must fit within on-chain size limits. Our optimization strategies:

- Utilize redeemer-indexing design pattern
- Reference scripts (Babbage) to avoid including code in transactions
- Leverage Aiken's built-in optimizations and efficient compilation

Table 16: Validator Sizes (ROSCA)

Validator	Size (bytes)
`sc1_mint_stt`	2494
`sc2_mint_cip68`	6902
`sc3_spend_contributions`	8494
`sc4_spend_collaterals`	4353

Table 17: Validator Sizes (ASCA)

Validator	Size (bytes)
`sc1_mint_stt`	2852
`sc2_mint_cip68`	6901
`sc3_spend_funds`	14742
`sc4_spend_membership`	12243
`sc5_stake_funds`*	2832

* without Delegation Management feature

8.4 User Experience Considerations

- **Wallet Compatibility**: Tested well with Eternl; should work with any CIP-30 compliant wallets
- **Internationalization**: UI available in English, French, Spanish, and Swahili (target markets)
- **Mobile Support**: Planned mobile app using Flutter

8.5 Monitoring and Analytics

Real-time dashboards track:

- Active groups and total TVL (Total Value Locked)
- Average contribution compliance
- Distribution commissions
- Loan approval rates
- Network-wide statistics (total users, total distributed, etc.)

9 Future Research Directions

9.1 Enhanced Randomness

Current PRNG using transaction hash is difficult to influence but could be enhanced with:

- Oracle integration for provable fair randomness
- Commit-reveal schemes for participant-contributed entropy
- Hybrid approaches combining multiple randomness sources

9.2 Cross-Chain Bridges

Enable participation from other blockchain ecosystems:

- Wrapped Ethereum or BSC
- Cross-chain voting via oracle attestations
- Multi-chain groups with atomic settlements

9.3 Credit Scoring

Integrate on-chain behavior analysis to:

- Assess borrower creditworthiness
- Suggest optimal loan terms
- Predict default risk for voters
- Personalize participation recommendations

9.4 Formal Verification

Complete formal proofs of:

- State machine safety properties
- Economic incentive compatibility
- Absence of value-draining bugs

Using tools like Agda, Isabelle/HOL, or K Framework.

10 Conclusion

The Akyba Protocol successfully translates centuries-old cooperative finance mechanisms into a deterministic, trustless, and transparent blockchain implementation. By leveraging the unique properties of Cardano's eUTxO model—deterministic execution and strong validation guarantees—Akyba provides a fully decentralized ROSCA and ASCA platform that preserves the behavioral economics of traditional tontines while eliminating their core vulnerabilities.

Through the combination of State Thread Tokens, CIP-68 reference architecture, and carefully designed validator logic, Akyba achieves:

- **Trustless Coordination**: No custodians or intermediaries required
- **Transparent Governance**: All decisions recorded immutably on-chain
- **Economic Security**: Collateral-based enforcement aligns incentives
- **Predictable Costs**: Deterministic transaction fees enhance user experience
- **Financial Inclusion**: Accessible to underbanked populations worldwide

As blockchain technology matures and adoption accelerates in emerging markets, protocols like Akyba demonstrate how decentralized systems can provide critical financial infrastructure for populations excluded from traditional banking. By encoding centuries of informal financial wisdom into cryptographic guarantees, Akyba represents a new paradigm for cooperative finance—one where trust is derived from mathematics rather than social relationships, and where transparency and fairness are ensured by code rather than hope. Future work will focus on formal verification, enhanced randomness sources, cross-chain interoperability, and integration with identity systems to further strengthen the protocol's security and usability. The successful deployment and operation of Akyba on Cardano mainnet will demonstrate the viability of eUTxO-based DeFi applications for real-world financial inclusion use cases.

Acknowledgments

We thank the Cardano Foundation, IOHK Research, the Aiken development team, and the broader Cardano smart contract developer community for their foundational work on the eUTxO model and modern tooling for Cardano. Special thanks to beta testers on Preview Testnet who provided invaluable feedback during protocol development. This research was supported by Oxalio Labs.

References

[1] Cardano Foundation, "CIP-68: On-chain token metadata via reference tokens," *Cardano Improvement Proposals*, 2022. [Online]. Available: https://cips.cardano.org/cips/cip68/

[2] M. M. T. Chakravarty et al., "The Extended UTXO Model," *Trusted Smart Contracts (WTSC)*, 2020. [Online]. Available: https://iohk.io/en/research/library/papers/the-extended-utxo-model/

[3] TxPipe, "Aiken: A modern smart contract platform for Cardano," 2023. [Online]. Available: https://aiken-lang.org/

[4] S. Thompson et al., "Marlowe: Financial Contracts on Blockchain," *IOHK Research*, 2018. [Online]. Available: https://iohk.io/en/research/library/papers/marlowe-financial-contracts-on-blockchain/

[5] M. Lelart, *La tontine: Pratique informelle d'épargne et de crédit*. Paris: Karthala, 1990.

[6] S. Ardener, "The Comparative Study of Rotating Credit Associations," *Journal of the Royal Anthropological Institute*, vol. 94, no. 2, pp. 201-229, 1964. [Online]. Available: https://www.jstor.org/stable/2844127

[7] F. J. A. Bouman, "Rotating and Accumulating Savings and Credit Associations: A Development Perspective," *World Development*, vol. 23, no. 3, pp. 371-384, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/0305750X9400141K

[8] C. Geertz, "The Rotating Credit Association: A 'Middle Rung' in Development," *Economic Development and Cultural Change*, vol. 10, no. 3, pp. 241-263, 1962. [Online]. Available: https://www.jstor.org/stable/1151982

[9] S. Rutherford, *The Poor and Their Money*. New Delhi: Oxford University Press, 2000. [Online]. Available: https://assets.publishing.service.gov.uk/media/57a08d0ae5274a27b20017a4/POORDOC1.pdf

[10] M. A. El-Gamal and B. G. Greaney, "Bank-Insured RoSCAs for Microfinance: Experimental Evidence from Egypt," *Journal of Economic Behavior & Organization*, vol. 95, pp. 162-174, 2013. [Online]. Available: https://www.adam-osman.com/wp-content/uploads/2020/01/3-JEBO-Bank-Insured-RoSCAs.pdf

[11] T. Besley, S. Coate, and G. Loury, "The Economics of Rotating Savings and Credit Associations," *The American Economic Review*, vol. 83, no. 4, pp. 792-810, 1993.

[12] S. Klonner, "Rotating Savings and Credit Associations When Participants Are Risk Averse," *International Economic Review*, vol. 44, no. 3, pp. 979-1005, 2003.

[13] S. Anderson and J.-M. Baland, "The Economics of Roscas and Intrahousehold Resource Allocation," *The Quarterly Journal of Economics*, vol. 117, no. 3, pp. 963-995, 2002.