RingCCT: confidential commit transactions and atomic swaps

Lorenzo Tucci

June 13, 2025

Contents

Narrative

- Universal atomic swaps allow atomic swaps across arbitrary pairs of chains which support ordinary transactions, in particular without requiring support of scripting, time-lock contracts, etc.
- This is appealing because most privacy chains (e.g. ZCash, Monero, MimbleWimble) do not support scripting.
- However, UAS requires both parties to solve TLPs, which are computationally intensive especially for lightweight clients.
- Moreover, it is tricky from an implementation perspective to properly set the difficulty level TLPs in UAS. For example, we identify a minor flaw ... and propose a fix.

Contributions

- We identify that a minimal chain functionality – commit transactions – suffices for achieving atomic swaps. Concretely, we propose a generic construction of an atomic swap protocol using only commit transactions and other basic functionalities of the chains. (To avoid dealing with UC, maybe we write this as an informal theorem?)
- We propose an extension of RingCT, the underlying transaction scheme of Monero, which allows to realise commit transactions in privacy-preserving cryptocurrencies. We propose a generic construction and show how to efficiently instantiate it over groups where discrete logarithm and other related problems are hard.
- We provide a prototype implementation of CommitTx-based atomic swaps

1 Introduction

An atomic cross-chain swap is a cryptographic protocol that enables two mutually distrustful parties to exchange assets residing on separate blockchains, ensuring the *atomicity* property: either both parties receive the corresponding asset from the other, or no transfer takes place. This guarantee must hold even in the presence of network delays or adversarial behavior, ensuring that neither party can cause a loss of funds for the other. To achieve this, atomic swap protocols rely on mechanisms that place each participants asset into a conditional transfer statelocked in such a way that it can be claimed by the counterparty upon satisfying specific conditions (e.g., revealing a secret or solving a puzzle), or refunded to the sender after a designated timeout.

The predominant approach for implementing atomic swaps is via Hash Time-Locked Contracts (HTLCs), which combine hash preimages with time-locks to enforce atomicity. While effective, HTLCs require on-chain scripting capabilities, a requirement that is incompatible with many privacy-focused cryptocurrencies such as Monero or ZCash. These systems intentionally restrict or eliminate scripting functionality to enhance privacy and efficiency, rendering HTLCs infeasible in their context.

This limitation poses a significant barrier to interoperability: despite the growing demand for cross-chain trading, privacy-preserving blockchains remain largely isolated due to the lack of compatible atomic swap mechanisms. Prior works, such as Universal Atomic Swaps (UAS), have explored protocols that replace hash-locks with time-lock puzzles (TLPs) to eliminate scripting assumptions. While theorically sound, TLP-based protocols face serious practical challenges. In particular, their security critically depends on potential difference in (linear) computational power, which may be of several orders of magnitude faster with certain hardware capabilities (e.g., ASICs). As a result, tuning puzzle difficulty to balance security and usability leads to protocols that are either impractical for honest users or insecure against adversaries with specialized hardware. This motivates the development of an atomic swap protocol that requires minimal on-chain functionality, making it suitable for privacy-focused blockchains like Monero and ZCash, which lack scripting capabilities, yet still supporting efficient and secure cross-chain asset transfers.

1.1 Our Contributions

In this work, we propose a new minimal blockchain functionality called *commit transactions* which enables supporting blockchains to perform atomic swap through minimal and efficient protocols. Our main contributions are outlined below.

Commit transactions. We propose *commit transactions*, a general-purpose blockchain functionality for expressing time-based conditional asset transfers. Commit accounts allow users to lock funds under dynamic ownership conditions, such as transitioning from shared control to unilateral recovery based on a predefined timeout. The functionality is compatible with both UTXO and account-based ledger models and can be integrated into public ledgers with minimal modification to the transaction verification logic, requiring no general-purpose scripting and incurring no additional transaction size overhead.

RingCCT (Ring Confidential Transaction). We extend the RingCT model to the commit transaction functionality. The scheme incorporates commit accounts with shared ownership semantics and time-conditional spending rules, while preserving the confidentiality and anonymity guarantees of standard RingCT. We present a concrete instantiation of RingCCT based on succinct non-interactive arguments, commitment schemes, and key-homomorphic pseudorandom functions.

Practical atomic swaps. Leveraging the commit transaction functionality, we design an efficient and secure generic protocol for secure cross-chain swaps. We further show how to construct practical atomic swap protocols that can be executed between any pair of ledgerswhether both are public blockchains, both are RingCCT-based, or a mix of the two.

1.2 Related Work

2 Preliminaries

General notation, e.g. security parameter, $[n]$, PPT, negligible, ...

2.1 Basic primitives

commitments, ZKP, ...

2.2 Non-Interactive Zero Knowledge Proofs

Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a NP-witness-relation with corresponding NP-language $\mathcal{L} := \{x : \exists w \text{ s.t. } R(x,w) = 1\}$

A non-interactive zero-knowledge proof (NIZK) system for R consist of the following algorithms:

- $\mathsf{cr} \leftarrow \mathsf{ZK}_{\mathcal{L}}.\mathsf{Setup}(1^\lambda)$ takes on input the security parameter, outputs a common reference string crs
- $\pi \leftarrow \mathsf{ZK}_{\mathcal{L}}.\mathsf{Pr}(\mathsf{crs}, x, w)$ takes on input the reference string crs, a statement x and a witness w, outputs a proof π
- $0/1 \leftarrow \mathsf{ZK}_{\mathcal{L}}.\mathsf{Vr}(\mathsf{crs}, x, \pi)$ takes on input the reference string crs, a statement x and a proof π. Outputs 1 if w is a witness for the statment x, 0 otherwise.

We require a NIZK system to be *zero-knowledge*, where the verifier does not learn more than the validity of the statement x, and *simulation sound* where it is hard for any prover to convince a verifier of an

invalid statement (chosen by the prover) even after having access to polynomially many simulated proofs for statements of his choosing.

2.3 Secure 2-Party Computation

A secure 2-party computa-tion (2PC) protocol allows two participating users P_0 and P_1 to securely compute some function f over their private inputs x_0 and x_1 respectively.

We require the standard *privacy* property, which states that the only information learned by the parties in the computation is that specified by the function output. We also require the standard security with aborts, where the adversary can decide whether the honest party will receive the output of the computation or not, and thus there are no assumptions on fairness or guaranteed output delivery.

2.4 Computational Assumptions

General setting of group-based crypto, implicit notation, assumptions that we need e.g. DLOG

3 Atomic Swaps and Existing Solutions

3.1 Existing solutions

3.1.1 Hash Time Lock Contracts

A Hash Time-Lock Contract (HTLC) is a contract that enables conditional payment based on the revelation of a cryptographic secret within a certain time window. Formally, an HTLC is characterized by a tuple $(\mathsf{amnt_a}, h, T, \mathsf{pk_0}, \mathsf{pk_1})$ where

- $\mathsf{amnt_a}$ denotes the amount of a assets to be exchanged
- h is the hash of a secret value r, i.e., $h = \mathcal{H}(r)$ for some cryptographic hash function $\mathcal{H}$
- T is a timelock parameter, typically a block height or timestamp, indicating the deadline after which funds can be refunded
- $\mathsf{pk_0}$ is the public key of the sender (who can reclaim the funds after the timeout).
- $\mathsf{pk_1}$ is the public key of the intended recipient (who can claim the funds upon presenting the correct preimage r before the timeout).

The HTLC transfers $\mathsf{amnt_a}$ to $\mathsf{pk_1}$ if invoked before timeout T with input value r such that $\mathcal{H}(r) = h$. If the contract is invoked after timeout T, it refunds the assets $\mathsf{amnt_a}$ to $\mathsf{pk_0}$ unconditionally.

Using HTLCs as a building block, an atomic swap protocol can be constructed as follows:
1) Alice chooses r, computes $h = \mathcal{H}(r)$, transfers $\mathsf{amnt_a}$ into an $(\mathsf{amnt_a}, h, T_0, \mathsf{pk_0}, \mathsf{pk_1})$ on blockchain $\mathbb{A}$ and sends h, T to Bob.
2) Bob finishes the setup of the exchange by choosing a time $T_1 < T_0$ and transferring his $\mathsf{amnt_b}$ assets into an HTLC$(\mathsf{amnt_b}, h, T_1, \mathsf{pk_{tx}}, \mathsf{pk_{rx}})$ on blockchain $\mathbb{B}$.

HTLCs serve as the core building block in many atomic swap protocols. A standard two-party protocol between Alice and Bob proceeds as follows:

Alice chooses a uniformly random secret value r, computes $h = \mathcal{H}(r)$, and initiates an HTLC on blockchain $\mathbb{A}$ locking $\mathsf{amnt_a}$ to Bob under the tuple $(\mathsf{amnt_a}, h, T_0, \mathsf{pkAlice}, \mathsf{pkBob})$. She then sends (h, T_0) to Bob.

Bob selects a smaller timeout $T_1 < T_0$ and sets up an HTLC on blockchain $\mathbb{B}$ locking his $\mathsf{amnt_b}$ to Alice under $(\mathsf{amnt_b}, h, T_1, \mathsf{pkBob}, \mathsf{pkAlice})$.

Alice redeems the funds from Bobs HTLC on $\mathbb{B}$ by revealing r. Since transactions on public blockchains are visible, Bob can then observe r on-chain and use it to redeem the funds from Alices HTLC on $\mathbb{A}$ before her timeout T_0.

This protocol ensures atomicity: either both parties receive their respective assets, or after the timeout, each party reclaims their original funds.

While HTLCs are widely used in practice, especially in early cross-chain systems and off-chain protocols such as the Lightning Network, they suffer from several critical limitations, especially when applied in a privacy-preserving or cross-paradigm setting.

Compatibility of hash functions HTLC-based atomic swaps require both blockchains involved to support the same hash function. For instance, if chain $\mathbb{A}$ supports SHA-256 and chain $\mathbb{B}$ supports Blake2, an HTLC constructed with a hash $h = \mathcal{H}(r)$ cannot be replicated on the second chain unless both parties can compute and verify the same preimage relation. One possible workaround is to have Alice compute $h = \mathcal{H}(r)$ and $h' = \mathcal{H}'(r)$ and publish a non-interactive zero-knowledge (NIZK) proof that both hashes are computed from the same preimage. However, this may increase protocol complexity and setup cost, and depends on the availability of NIZK-friendly primitives on both chains. Since the same preimage r is reused on both chains, an external observer can trivially link the two transactions, breaking the unlinkability that privacy-preserving systems aim to provide. A workaroundsuch as committing to a shifted value $r+r'$ with an accompanying NIZK proofadds cryptographic overhead and requires careful implementation to avoid privacy leaks.

(In)compatibility with private ledgers Many privacy-preserving cryptocurrencies, such as Monero or Zcash, do not support expressive scripting or global hash preimage verifiability. Even when HTLCs are theoretically implementable (e.g., in a limited form over RingCT), their transaction structure becomes easily distinguishable from standard private transfers, harming plausible deniability. In such cases, the protocol becomes asymmetric: the public-chain participant must move first, revealing the preimage, which the private-chain user can then observe off-chainthis violates the symmetry typically desired in fair exchange protocols.

Miner incentives attacks HTLCs may also suffer from incentive misalignments, particularly in adversarial mining environments. Miners observing the hash preimage during the redemption phase may attempt to front-run or extract value, especially when the reward from a successful claim is higher than the block reward or other fees. These risks have been documented in Kolluri et al., 2022, which explores protocol designs vulnerable to such "griefing" attacks in HTLC settings.

3.1.2 Universal Atomic Swaps

- this needs to be a little rewritten, pointing out on the impracticaly of the protocol due to ASICs
- also mention the "bug" of the proposed protocol due to not considering chain's confirmation time (might be used as motivation for our blockchain interface?)

Thyagarajan et al. [**uas**] introduced one of the first atomic swap protocols substituting blockchain timelocks with a cryptographic primitive. The core building block utilized are Verifiable Timed Signatures (VTS) [**vts**], which lets a user generate a timed commitment C of a signature σ on a message m under a public key pk. The commitment C must hide the signature σ for time T and producing a proof π that C contains a valid signature σ. This ensures that σ can be publicly recovered in time T by anyone who solves the computational puzzle. We note that a similar construction called Verifiable Timed Discretelog (VTD) allows to commit on a dlog value instead of a signature, and can be alternatively used in the protocol.

Let P_0 and P_1 be two parties where P_0 wants to exchange $\mathsf{amnt_a}$ on blockchain $\mathbb{A}$ from their address $\mathsf{pk}_{\mathsf{init}(\mathbb{A})}$ for $\mathsf{amnt_b}$ on blockchain $\mathbb{B}$ to $\mathsf{pk}_{\mathsf{swp}(\mathbb{B})}$ and vice-versa for P_1.

In the setup phase of the protocol, the parties run a 2PC protocol to setup two freeze addresses on the respective chains $\mathsf{pk}_{\mathsf{frz}(\mathbb{A})}$ and $\mathsf{pk}_{\mathsf{frz}(\mathbb{B})}$, where each party posseses one share of the respective secret keys, e.g. $\mathsf{sk}_{\mathsf{frz}(\mathbb{A})} := \mathsf{sk}_{\mathsf{frz0}} \oplus \mathsf{sk}_{\mathsf{frz1}}$.
Now the parties create a refund transaction transferring back the assets in case of timeout, for P_0 we have $\mathsf{tx}_{\mathsf{rfnd}(\mathbb{A})}$ refunding amnt_a from $\mathsf{pk}_{\mathsf{frz}(\mathbb{A})}$ to $\mathsf{pk}_{\mathsf{init}(\mathbb{A})}$ and similarly for P_1 $\mathsf{tx}_{\mathsf{rfnd}(\mathbb{B})}$.
Each party generates a timed commitment on the signature of the counterparty's refund transaction, where P_0 receives a VTS with commitment C_0 and timeout $T_0 = T_1 + \Delta$ and P_1 receives a VTS with commitment C_1 and timeout T_1. Once both VTS commitment are verified the parties proceed to transfer the assets to the freeze addresses, assured to retrieve the refund transaction signatures after force opening the commitments with timeouts T_0 and T_1.

In the subsequential lock phase, parties first initialize the swap transactions $\mathsf{tx}_{\mathsf{swp}}$ transferring amnt from $\mathsf{pk}_{\mathsf{frz}}$ to $\mathsf{pk}_{\mathsf{swp}}$ for the respective chains. They then compute via 2PC $\mathsf{lk} := \sigma_{\mathsf{swp}(\mathbb{A})} \oplus \mathcal{H}(\sigma_{\mathsf{swp}(\mathbb{B})})$, where P_0 receives $\sigma_{\mathsf{swp}(\mathbb{B})}$ and P_1 receives lk. When P_0 publishes $\mathsf{tx}_{\mathsf{swp}(\mathbb{B})}$ together with $\sigma_{\mathsf{swp}(\mathbb{B})}$, P_1 can unmask lk by computing $\mathcal{H}(\sigma_{\mathsf{swp}(\mathbb{B})})$ to retrieve $\sigma_{\mathsf{swp}(\mathbb{A})}$ and publish $\mathsf{tx}_{\mathsf{swp}(\mathbb{A})}$.

If P_0 fails to publish $\mathsf{tx}_{\mathsf{swp}(\mathbb{B})}$ before T_1, P_1 will publish the refund transaction $\mathsf{tx}_{\mathsf{rfnd}(\mathbb{B})}$ and similarly for P_0 if P_1 timeouts during the protocol execution.

Note that parties must also take into account potential differences in the computational power available for force opening the VTS commitments. This prevent scenarios where one party force opens its VTS commitments earlier than expected, potentially stealing the other party's assets during the swap lock or complete phase. Therefore, Δ (such that T0 = T1 + Δ) must be large enough to tolerate time differences in opening the VTS commitments.

4 Commit Transactions and Atomic Swaps

4.1 Commit Transactions

The commit transaction primitive offers a lightweight and expressive mechanism for implementing conditional, time-sensitive asset transfers directly on-chain, without requiring a general-purpose scripting environment. This functionality enables users to lock funds under epoch-dependent spending conditions, supporting contract patterns such as atomic swaps, escrows, and delayed claimsall of which rely on time-based control over asset ownership.
We note that providing a fully universal definition of a commit transaction primitive is challenging due to the diversity of transaction models across blockchain systems. In particular, fundamental differences between UTXO-based (e.g., Bitcoin, MimbleWimble) and account-based (e.g., Ethereum, ZCash Sapling) systems lead to distinct transaction semantics and interface requirements. As a result, a single formalization of commit transactions that universaly applies to all transaction schemes would either be overly abstract or would need to be tailored closely to each specific model. Nevertheless, the underlying concept of commit transactions - namely, timeout conditional asset ownership remains simple and expressive. In practice, this functionality can be readily instantiated in various ledger models with modest modifications to fit the specifics of the transaction scheme and primitives in use.

The functionality of a commit transaction can be logically divided into two phases: Commit Phase and Reveal Phase, described as follows:
Commit Phase: The user locks a given amount of coins into a special account defined by a commitment to:

- A main secret key sk_0
- A set of auxiliary secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_k$

- Two (or more) index sets $I_0, I_1 \subseteq [k]$ which define the required authorized key combinations for spending the funds
- A time threshold T, denoting an epoch after which the ownership of the accounts transitions.

Reveal Phase: Depending on whether a transaction is issued before or after the epoch threshold T, different subsets of auxiliary keys are required:

- Before time T: The account can be spent using sk_0 and the auxiliary keys indexed by I_0
- After time T: The spending requires sk_0 and keys indexed by I_1

This time-dependent control structure enables the definition of disjoint execution paths, ensuring that only authorized parties can redeem the funds within their designated time windows. For example, in an atomic swap protocol, one party may claim the funds by providing the required keys before a deadline, while the other party regains control if the swap fails and the deadline elapses.

4.2 Commit-Transaction-based Atomic Swaps

In this section we construct a protocol that realizes atomic swaps on a generic blockchain interface supporting commit transactions.

- motivate use of this blockchain definition instead of UC-style functionality
- explain how to set T_0, T_1 with respect of confirmation time

4.2.1 Blockchain syntax

Definition 1. *A blockchain system supporting confidential transactions is defined by the following sets of the algorithms*

$$(\mathsf{TxPub}, \mathsf{TxGen}, \mathsf{CommitTx}, \mathsf{TxVf}, \mathsf{GetState}, \mathsf{TimeExt})$$

- $\mathbf{0/1} \leftarrow \mathbf{TxPub}(\mathsf{tx})$*: publishes the transaction* tx *on the blockchain. Outputs 1 if the transaction is accepted, 0 otherwise.*
- $\mathbf{tx} \leftarrow \mathbf{TxGen}(\mathsf{st}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in\mathsf{SC}}, \mathsf{pk}_{\mathsf{TG}}, \mathsf{amnt})$*: generates a signed transaction transferring an amount* amnt *from a source account associated to the keypairs in the set* $\mathsf{SC} = \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in\mathsf{SC}}$ *to the target public key* $\mathsf{pk}_{\mathsf{TG}}$*, based on the current state* st*.*
- $\mathbf{tx} \leftarrow \mathbf{CommitTx}(\mathsf{st}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in\mathsf{SC}}, \mathsf{pk}_{\mathsf{TG}}, \{\mathsf{pk}_i\}_{i\in I_0}, \{\mathsf{pk}_i\}_{i\in I_1 1}, T, \mathsf{amnt})$*: produces a commit transaction transferring* amnt *from a source account associated to the keypairs in the set* $\mathsf{SC} = \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in\mathsf{SC}}$ *to a commit-type account defined by a main public key* $\mathsf{pk}_{\mathsf{TG}}$*, auxiliary key sets* $I_0 = \{\mathsf{pk}_i\}_{i\in I_0}$ *and* $I_1 = \{\mathsf{pk}_i\}_{i\in I_1}$*, and a timeout parameter* T*.*
- $\mathbf{0/1} \leftarrow \mathbf{TxVf}(\mathsf{st}, \mathsf{tx})$*: Verifies whether a transaction* tx *is valid under the given blockchain state* st *and its encoded epoch, which may be extracted using* $\mathsf{TimeExt}$*. If the source account of the transaction is a commit-type account, it requires that the secret keys corresponding to* pk_T *and either one of the auxiliary sets,* $\{\mathsf{pk}_i\}_{i\in I_0}$ *or* $I_1 = \{\mathsf{pk}_i\}_{i\in I_1}$ *are included in the set* SC *based on defined timeout* T*. Returns 1 if the transaction is valid, 0 otherwise.*
- $\mathsf{st} \leftarrow \mathbf{GetState}$*: Returns the current state* st *from the global system state (consensus), including account records, verified transactions, and the current epoch.*
- $\mathsf{time} \leftarrow \mathbf{TimeExt}$*: extracts the epoch from the state.*

4.2.2 Construction

Notation. In order to encode the concurrent execution of routines in an asynchronous setting, we use the following notation in the protocol's pseudocode definition.

- **wait** $\mathsf{fn}(...)$ - Suspends execution until the execution of the algoritm $\mathsf{fn}(...)$ terminates. If $\mathsf{fn}(...)$ returns $\perp$, the current execution block aborts and returns $\perp$.
- **wait** $\{...\}$ - Enforces the **wait** operation to all asynchonous routines inside the execution block.
- **assert** $\{...\}$ - Verifies that enclosed expression or routine returns 1. If not, the current block terminates and returns $\perp$.
- **select** $\{...\}$ - Concurrently runs multiple asynchronous **wait** routines in the block and returns the value of the first routine that completes successfully.

All routines that interact with network channelssuch as accessing the blockchain interface or executing a two-party computation (2PC) with another participantare treated as asynchronous. This reflects the fact that such operations may incur unpredictable delays and cannot be assumed to complete within a fixed timeframe.
In the protocol specification, variables and routines that are specific to a particular blockchain $\mathbb{B}$ are annotated with a subscript to distinguish their context, unless otherwise clear. For example, a public key belonging to chain $\mathbb{B}$ is denoted by $\mathsf{pk}_{(\mathbb{B})}$.
Parties may communicate over an unreliable and unauthenticated channel, with no assumptions regarding security or message delivery guarantees. Communication is modeled using the primitives send and $\mathsf{receive}$, which respectively transmit and await data over the channel.

Setup. Let $\mathbb{A}$ and $\mathbb{B}$ denote two blockchains that support confidential constructions, as defined previously. Consider two parties, P_0 and P_1, who wish to perform a cross-chain asset exchange: P_0 aims to swap $\mathsf{amnt}_{\mathbb{A}}$ units of currency on $\mathbb{A}$ in exchange for $\mathsf{amnt}_{\mathbb{B}}$ units on $\mathbb{B}$ from P_1, and vice versa.
Each party generates keypairs for both chains. Specifically, P_0 generates $(\mathsf{pk}_i, \mathsf{sk}_i)$ on chain $\mathbb{A}$ and $(\mathsf{pk}_s, \mathsf{sk}_s)$ on chain $\mathbb{B}$, while P_1 generates $(\mathsf{pk}_i, \mathsf{sk}_i)$ on $\mathbb{B}$ and $(\mathsf{pk}_s, \mathsf{sk}_s)$ on $\mathbb{A}$.
The parties agree on a common set of global parameters: timeouts T_0 and T_1, and transfer amounts $\mathsf{amnt}_{\mathbb{A}}$ and $\mathsf{amnt}_{\mathbb{B}}$. Given this setup, both parties proceed to execute the protocol described in **??**, using the global parameters as public input and their respective keypairs as private input.

Key generation and commit phase. Each party begins by generating three cryptographic key pairs, each serving a distinct role in the protocol. The pair $(\mathsf{sk}_m, \mathsf{pk}_m)$ is designated as the main keypair, associated with the control of a commit-type account that will eventually hold the locked assets. The second pair, $(\mathsf{sk}_r, \mathsf{pk}_r)$ functions as a recovery keypair, enabling the party to reclaim funds in case the protocol fails to complete within the allotted timeout. The third pair, $(\mathsf{sk}_c, \mathsf{pk}_c)$ is the commitment keypair, which is owned by the respective counterparty in order to allow for a joint ownership of the commit account. Party P_0 generates its main and recovery keypairs on blockchain $\mathbb{B}$. Party P_1 follows the symmetric strategy: it generates its main and recovery keypairs on blockchain $\mathbb{B}$, and its commitment keypair on blockchain $\mathbb{A}$. Following key generation, the parties exchange their respective public commitment keys over an off-chain channel.
Using this setup, P_0 locally executes, with respect to chain $\mathbb{A}$

$$\mathsf{CommitTx}(\mathsf{st}, \{(\mathsf{pk_i}, \mathsf{sk_i})\}, \mathsf{pk_m}, \{\mathsf{pk_c}\}, \{\mathsf{pk_r}\}, T_0, \mathsf{amnt})$$

, where $(\mathsf{pk_i}, \mathsf{sk_i})$ is the party's source keypair, amnt is the amount to be exchanged with respect to $\mathbb{A}$, T_0 is the agreed timeout parameter, $\mathsf{pk_m}$ and $\mathsf{pk_r}$ are the main and recovery public key, and $\mathsf{pk_c}$ is the commitment public key received by the counterparty. P_1 then symmetrically executes $\mathsf{CommitTx}$ with respect to chain $\mathbb{B}$ and using timeout T_1, and both parties publish the transactions tx_r through the **TxPub**.

Swap and refund phase. The parties now proceed to concurrently write two routines corresponding to the swap and refund mechanism. The refund routine checks whether T_0 or T_1 (respectively for P_0 and P_1

are timed out, and if so generate and publish the refund transaction using the master and recovery key pair as source $\mathsf{SC} := \{(\mathsf{pk_m}, \mathsf{sk_m}), (\mathsf{pk_r}, \mathsf{sk_r})\}$ paying back amnt to $\mathsf{pk_i}$.
The swap routine send the previously published transactions initialized the commit accounts tx_r to the respective counterparty, and they both verify that the transactions are valid through the algorithm TxVf and that the the $\mathsf{pk_c}$ and amnt are set correctly. The parties now proceed to run the 2PC protocol Γ_{Swap} as defined in **??**. After publishing the commit transactions, both parties proceed to concurrently execute two protocol routines that implement the refund and swap mechanisms. These routines are designed to ensure liveness and atomicity of the asset exchange in the presence of network delays or adversarial behavior. The refund routine serves as a timeout-based recovery path. Each party locally monitors the blockchain state and checks whether the corresponding timeout parameter or has expired. Upon detecting a timeout, the party generates and publishes a refund transaction using its main and recovery keypairs as signing authorities, i.e., it sets as source keypairs $\mathsf{SC} := \{(\mathsf{pk_m}, \mathsf{sk_m}), (\mathsf{pk_r}, \mathsf{sk_r})\}$. This transaction transfers the originally committed amount amnt back to the inital public key pair pk_i, ensuring that funds are reclaimed securely if the protocol fails to complete.
In parallel, each party initiates the swap routine, which coordinates the completion of the atomic asset exchange. The parties begin by sending the previously published commit transactions tx_r, which initialize the commit-type accounts, to their respective counterparties over an off-chain channel. Upon receipt, each party verifies the validity of the received transaction using the verification algorithm. Additionally, they confirm that the embedded commitment public key $\mathsf{pk_c}$ and the committed amount amnt match the expected values agreed upon during setup. If the validation succeeds, the parties then execute the two-party computation protocol Γ_{Swap}, as defined in **??**. The 2PC protocol Γ_{Swap} carries out a secure joint computation between the two parties to generate the swap transactions that transfer assets from the respective commit-type accounts to the target settlement public keys pk_s on the respective chains. As part of its execution, the protocol verifies the correctness and consistency of each partys inputsnamely, their key material, local state views, and agreed transfer amountsbefore proceeding with transaction construction. To enforce atomicity, the protocol introduces a commitment mechanism that prevents either party from unilaterally finalizing the swap. Specifically, it constructs a cryptographic lock by computing

$$\mathsf{lk} := \mathcal{H}(\mathsf{tx}_{s,\mathbb{B}}) \oplus \mathsf{tx}_{\mathsf{s},\mathbb{A}}$$

, where $\mathsf{tx}_{s,\mathbb{B}}$ and $\mathsf{tx}_{s,\mathbb{A}}$ are the swap transactions of parties P_0 and P_1. Here, $\mathcal{H}(\cdot)$ denotes a cryptographic hash function, and $\oplus$ indicates bitwise exclusive-or. The result is a binding encoding that ties the two transactions together such that the publication of one enables the recovery of the other. Finally, Γ_{Swap} outputs the lock lk to party P_1 and the transaction $\mathsf{tx}_{s,\mathbb{B}}$ to party P_0. Upon receiving $\mathsf{tx}_{s,\mathbb{B}}$, party P_0 proceeds to publish the transaction directly on the target chain $\mathsf{tx}_{s,\mathbb{B}}$. Meanwhile, the counterparty P_1 enters a monitoring phase, during which it continuously polls the state of chain $\mathbb{B}$ to detect the appearence of the expected swap transaction $\mathsf{tx}_{s,\mathbb{B}}$.
Once $\mathsf{tx}_{s,\mathbb{B}}$ is observed on-chain, P_1 leverages the leverages the previously received lock value lk to recover the corresponding transaction swap by computing $\mathsf{tx}_{s,\mathbb{A}} := \mathsf{lk} \oplus \mathcal{H}(\mathsf{tx}_{s,\mathbb{B}})$, which may now be published to complete complete the swap.
The use of the XOR-locked encoding guarantees that P_1 can finalize its transaction only after observing the publication of P_0's transaction on $\mathbb{B}$.
This mechanism enforces a strict dependency between the two on-chain actions, thereby preserving the atomicity of the swap. Furthermore, the protocol is resilient to failures or delays during the swap phase. If any assertion fails, any subroutine returns an error, or any asynchronous operation stalls indefinitelysuch as unresponsiveness from a counterparty or delay in transaction propagationeach party is programmed to invoke the refund mechanism once the respective timeout T_0 or T_1 elapses. This design guarantees that, even in the presence of faults or adversarial behavior, no party suffers financial loss, and the protocol atomicity is mantained.

$P_0((\mathsf{pk_m}, \mathsf{sk_m})_{\mathbb{A}}, (\mathsf{pk_c}, \mathsf{sk_c})_{\mathbb{B}}, \mathsf{pk}_{\mathsf{s},\mathbb{B}})$	$P_1((\mathsf{pk_m}, \mathsf{sk_m})_{\mathbb{B}}, (\mathsf{pk_c}, \mathsf{sk_c})_{\mathbb{A}}, \mathsf{pk}_{\mathsf{s},\mathbb{A}})$
assert $(\mathsf{st}_{\mathbb{B}}, \mathsf{st}_{\mathbb{A}}, \mathsf{amnt}_{\mathbb{B}}, \mathsf{amnt}_{\mathbb{A}})^0 = (\mathsf{st}_{\mathbb{B}}, \mathsf{st}_{\mathbb{A}}, \mathsf{amnt}_{\mathbb{B}}, \mathsf{amnt}_{\mathbb{A}})^1$	
$\mathsf{tx}_{\mathsf{s},\mathbb{B}} := \mathsf{TxGen}_{\mathbb{B}}(\mathsf{st}, \{(\mathsf{pk}_{\mathsf{m}}^1, \mathsf{sk}_{\mathsf{m}}^1), (\mathsf{pk}_{\mathsf{c}}^0, \mathsf{sk}_{\mathsf{c}}^0)\}, \mathsf{pk}_{\mathsf{s}}^0, \mathsf{amnt})$	
assert $\mathsf{TxVf}_{\mathbb{B}}(\mathsf{st}, \mathsf{tx}_{\mathsf{s}})$	
$\mathsf{tx}_{\mathsf{s},\mathbb{A}} := \mathsf{TxGen}_{\mathbb{A}}(\mathsf{st}, \{(\mathsf{pk}_{\mathsf{m}}^0, \mathsf{sk}_{\mathsf{m}}^0), (\mathsf{pk}_{\mathsf{c}}^1, \mathsf{sk}_{\mathsf{c}}^1)\}, \mathsf{pk}_{\mathsf{s}}^1, \mathsf{amnt})$	
assert $\mathsf{TxVf}_{\mathbb{A}}(\mathsf{st}, \mathsf{tx}_{\mathsf{s}})$	
$\mathsf{lk} := \mathcal{H}(\mathsf{tx}_{\mathsf{s},\mathbb{B}}) \oplus \mathsf{tx}_{\mathsf{s},\mathbb{A}}$	
output lk **to** P_1	
output $\mathsf{tx}_{\mathsf{s},\mathbb{B}}$ **to** P_0	

Figure 1: Protocol definition of 2PC $\Gamma_{\mathsf{CommitTx}}$

Party input $(\mathsf{pk_i}, \mathsf{sk_i})_{\mathbb{A}}, (\mathsf{pk_s}, \mathsf{sk_s})_{\mathbb{B}}$

$(\mathsf{sk_m}, \mathsf{pk_m})^0 \leftarrow \mathsf{KGen}_{\mathbb{A}}(\mathsf{pp})$

$(\mathsf{sk_r}, \mathsf{pk_r})^0 \leftarrow \mathsf{KGen}_{\mathbb{A}}(\mathsf{pp})$

$(\mathsf{sk_c}, \mathsf{pk_c})^0 \leftarrow \mathsf{KGen}_{\mathbb{B}}(\mathsf{pp})$

$\mathsf{send}(\mathsf{pk}^0_{\mathsf{c},\mathbb{B}})$

$\mathsf{pk}^1_{\mathsf{c},\mathbb{A}} \leftarrow \mathsf{receive}$

$\mathsf{tx}_{\mathsf{r},\mathbb{A}} \leftarrow \mathsf{CommitTx}_{\mathbb{A}}(\mathsf{st}, \{(\mathsf{pk_i}, \mathsf{sk_i})^0\}, \mathsf{pk}^0_{\mathsf{m}}, \{\mathsf{pk}^1_{\mathsf{c}}\}, \{\mathsf{pk}^1_{\mathsf{r}}\}, T_0, \mathsf{amnt})$

$\mathsf{TxPub}_{\mathbb{A}}(\mathsf{tx_r})$

select {

  wait {

    do $\mathsf{st} \leftarrow \mathsf{GetState}_{\mathbb{A}}$

    while $\mathsf{TimeExt}_{\mathbb{A}}(\mathsf{st}) < T_0$

    $\mathsf{tx_r} \leftarrow \mathsf{TxGen}_{\mathbb{A}}(\mathsf{st}, \{(\mathsf{pk_m}, \mathsf{sk_m})^0, (\mathsf{pk_r}, \mathsf{sk_r})^0\}, \mathsf{pk}^0_{\mathsf{i}}, \mathsf{amnt})$

    $\mathsf{TxPub}_{\mathbb{A}}(\mathsf{tx_r})$

  }

  wait {

    $\mathsf{send}(\mathsf{tx}_{\mathsf{r},\mathbb{A}})$

    $\mathsf{tx}_{\mathsf{r},\mathbb{B}} \leftarrow \mathsf{receive}$

    assert $\mathsf{TxVf}_{\mathbb{B}}(\mathsf{tx}_{\mathbb{B}})$

    assert $(\mathsf{pk}^0_{\mathsf{c},\mathbb{B}}, \mathsf{amnt}_{\mathbb{B}}) \in \mathsf{tx}_{\mathsf{r},\mathbb{B}}$

    $\mathsf{tx}_{\mathsf{s},\mathbb{B}} \leftarrow \Gamma.\mathsf{Swap}(\mathsf{pk}^0_{\mathsf{m},\mathbb{A}}, \mathsf{pk}^0_{\mathsf{r},\mathbb{A}}, \mathsf{pk}^0_{\mathsf{c},\mathbb{B}})$

    assert $\mathsf{TxPub}_{\mathbb{B}}(\mathsf{tx_s})$

  }

}

Party input $(\mathsf{pk_i}, \mathsf{sk_i})_{\mathbb{B}}, (\mathsf{pk_s}, \mathsf{sk_s})_{\mathbb{A}}$

$(\mathsf{sk_m}, \mathsf{pk_m})^1, \leftarrow \mathsf{KGen}_{\mathbb{B}}(\mathsf{pp})$

$(\mathsf{sk_r}, \mathsf{pk_r})^1 \leftarrow \mathsf{KGen}_{\mathbb{B}}(\mathsf{pp})$

$(\mathsf{sk_c}, \mathsf{pk_c})^1 \leftarrow \mathsf{KGen}_{\mathbb{A}}(\mathsf{pp})$

$\mathsf{send}(\mathsf{pk}^1_{\mathsf{c},\mathbb{A}})$

$\mathsf{pk}^0_{\mathsf{c},\mathbb{B}} \leftarrow \mathsf{receive}$

$\mathsf{tx}_{\mathbb{B}} \leftarrow \mathsf{CommitTx}_{\mathbb{B}}(\mathsf{st}, \{(\mathsf{pk_i}, \mathsf{sk_i})^1\}, \mathsf{pk}^1_{\mathsf{m}}, \{\mathsf{pk}^0_{\mathsf{c}}\}, \{\mathsf{pk}^1_{\mathsf{r}}\}, T_1, \mathsf{amnt})$

$\mathsf{TxPub}_{\mathbb{B}}(\mathsf{tx}_{\mathbb{B}})$

select {

  wait {

    do $\mathsf{st} \leftarrow \mathsf{GetState}_{\mathbb{B}}$

    while $\mathsf{TimeExt}_{\mathbb{B}}(\mathsf{st}) < T_1$

    $\mathsf{tx_r} \leftarrow \mathsf{TxGen}_{\mathbb{B}}(\mathsf{st}, \{(\mathsf{pk_m}, \mathsf{sk_m})^1, (\mathsf{pk_r}, \mathsf{sk_c})^1\}, \mathsf{pk}^1_{\mathsf{i}}, \mathsf{amnt})$

    $\mathsf{TxPub}_{\mathbb{B}}(\mathsf{tx_r})$

  }

  wait {

    $\mathsf{send}(\mathsf{tx}_{\mathsf{r},\mathbb{B}})$

    $\mathsf{tx}_{\mathsf{r},\mathbb{A}} \leftarrow \mathsf{receive}$

    assert $\mathsf{TxVf}_{\mathbb{A}}(\mathsf{tx_r})$

    assert $(\mathsf{pk}^1_{\mathsf{c},\mathbb{A}}\, \mathsf{amnt}_{\mathbb{A}}) \in \mathsf{tx}_{\mathsf{r},\mathbb{A}}$

    $\mathsf{lk} \leftarrow \Gamma.\mathsf{Swap}(\mathsf{pk}^1_{\mathsf{m},\mathbb{B}}, \mathsf{pk}^1_{\mathsf{r},\mathbb{B}}, \mathsf{pk}^1_{\mathsf{c},\mathbb{A}})$

    do $\mathsf{st} \leftarrow \mathsf{GetState}_{\mathbb{B}}$

    while $\nexists\, \mathsf{tx} \in \mathsf{st} \mid (\mathsf{pk}^1_{\mathsf{c},\mathbb{B}}, \mathsf{pk}^0_{\mathsf{m},\mathbb{B}}) \in \mathsf{tx}$

    $\mathsf{tx}_{\mathsf{s},\mathbb{B}} := \mathsf{tx} \in \mathsf{st} \mid (\mathsf{pk}^1_{\mathsf{c},\mathbb{B}}, \mathsf{pk}^0_{\mathsf{m},\mathbb{B}}) \in \mathsf{tx}$

    $\mathsf{tx}_{\mathsf{s},\mathbb{A}} := \mathsf{lk} \oplus \mathsf{tx}_{\mathsf{s},\mathbb{B}}$

    assert $\mathsf{TxPub}_{\mathbb{A}}(\mathsf{tx_s})$

  }

}

Figure 2: Full protocol execution for P_0 and P_1, respectively left and right

4.3 Comparison with HTLC-based Atomic Swaps

- Why chains might want to support commit transactions but not "hash time lock contracts (HTLC)"?
- Commit transaction is local functionality, cannot be directly compare

5 RingCCT: Ring confidential commit transaction

- need to swap to a PRF scheme, using only two master keys

We present an extension of RingCT called ring confidential commit transactions (RingCCT), which introduces an additional account abstraction enabling timeout-dependent ownership logic for accounts jointly created by two parties. The system state now mantains, in addition to the the list of all existing accounts and associated tags, an integer-valued epoch counter which is incremented upon each succesful verification. Similar to RingCT, each account is associated with a commitment encoding an amount and the corresponding set of public keys. However, the commited data now additionaly encodes a bit indicating the account type and a (possibly zero-valued) timeout epoch parameter, which governs conditional control over the account. If the account type is standard, the timeout is set to zero and the system behaves analogously as before. Otherwise, the account type is said to be *commit-based*, and its ownership is jointly held by two distinct users until the system epoch reaches the account-specified timeout. Upon expiration, the ownership reverts to a single designated user.

5.1 Syntax

System setup and state. A RingCCT system is initialized by running the setup algorithm Setup to generate public parameters and the initial state st. Users may invoke the key genation algorithm KGen to obtain a master key pair $\mathsf{mpk}, \mathsf{msk}$. The set of all accounts in the systems, denoted by $\mathsf{AC_U} = \{\mathsf{ac}_i\}_{i \in U}$ (where U stands for universe), can be extracted from the system state st via the algorithm StExt. The state also encodes an integer-valued epoch counter which can be extracted with the algorithm $\mathsf{TimeExt}$.

Accounts. Each account ac comprises a commitment co to some account data accd, along with a tuple of public keys $\mathsf{pks} = (\mathsf{spk}, \mathsf{tpk}, \mathsf{rpk})$, each corresponding to the secret keys $\mathsf{sks} = (\mathsf{ssk}, \mathsf{tsk}, \mathsf{rsk})$. The account data accd encodes an amount $a \in \mathbb{Z}$, a timeout epoch $\mathsf{tout} \in \mathbb{N}$, and the account type identified by the bit $b \in \{0, 1\}$. If $b = 0$, the account is said to be *standard* and is exclusively controlled by a single master secret key msk. In this case, only the key pair $(\mathsf{spk}, \mathsf{ssk})$ is populated, with the remaining keys in pks set to $\perp$. If $b = 1$, the account is classified as *commit-based* and is associated to two distinct master keys, possibily belonging to distinct users. Prior to epoch tout, the account is jointly controlled by the secret keys ssk and tsk, where tsk is associated to a distinct master secret key. After the tout, control over the account transitions to rsk exclusively, which is bound to the same master key pair as ssk.

Transaction generation. Transactions are generated via the algorithm TxGen, which takes as input the system state st, an index set $R \subseteq U$ called the ring, a set of source account information $\mathcal{S} = \{\mathsf{sks}_i, r_i, \mathsf{accd}_i\}_{i \in S}$ for some $S \subseteq R$, a set of target account data $\mathcal{T} = \{\mathsf{mpks}_i, \mathsf{accd}'_i\}_{i \in T}$, and a predicate P over source and target amounts (e.g.,enforcing asset conservation). Each source account $\mathsf{ac}_i := (\mathsf{accd}_i, \mathsf{co}_i)$, for $i \in S$, belongs to the account collection $\mathsf{AC}U$ and holds an amount a_i encoded in $\mathsf{accd}_i = (a_i, \mathsf{tout}_i, b_i)$. The transaction specifies that the amounts $a_i i \in S$ are to be transferred to a set of target accounts $\mathsf{ac}'ii \in T$, where each ac'_i is associated with a received amount a'_i stored in $\mathsf{accd}'i$. Correctness of the transaction requires that the predicate $P(a_i i \in S, a'ii \in T) = 1$ holds, ensuring the intended relation between source and target assets. The set of source and target accounts may contain a mix of standard and commit-based types. However, to maintain consistency, all commit-based source accounts must share the same epoch timeout tout, since a transaction may only encode a single such timeout. The TxGen additionally generates the set of tokens $\mathsf{TK} = \{\mathsf{tx}_i\}_{i \in T}$ corresponding to the target accounts $\{\mathsf{ac}'_i\}_{i \in T}$. These tokens may be used in the key derivation algorithm KDer to derive the associated secret keys and account data tuple necessary for future transactions. Tokens are assumed to be securely exchanged to the respective owners of the target accounts through a secure channel. Alternatively each token tk_i may be encrypted under the recipient's master public key and appended to the corresponding ac'_i to enable future recovery.

Verification. Once a transaction is published, each user may run the verification algorithm TimeVf deciding whether the transaction is valid with respect to the current state st and the current epoch $\mathsf{time} = \mathsf{TimeExt}(\mathsf{st})$. If the verification succeds, the verification outputs and updated state st'.
An auxiliary verification algorithm Vf may be used to check the validity of a transaction with respect to an arbitrary epoch time. The correctness and integrity of account usage within a transaction are verified using the auxiliary algorithms SrcChk and TgtChk.

Definition 2 (Ring Confidential Commit Transactions (RingCCT))**.** *A RingCCT scheme consists of the PPT algorithms* $\mathsf{Setup}, \mathsf{KGen}, \mathsf{KDer}, \mathsf{TxGen}, \mathsf{Vf}, \mathsf{TimeVf}, \mathsf{StExt}, \mathsf{TxExt}, \mathsf{TimeExt}, \mathsf{ExtAccTout}, \mathsf{EvalTags}, \mathsf{TimedOut}, \mathsf{SrcChk}, \mathsf{TgtChk}$ *whose interfaces are defined as follows.*

- $(\mathsf{pp}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda)$*: the setup algorithm generates the public parameters* st *and an initial global state* st*.*
- $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{KGen}(\mathsf{pp})$*: the key generation algorithm generates a master public key* mpk *and a matching secret key* msk*.*
- $(\mathsf{sk}, \mathsf{accd}) \leftarrow \mathsf{KDer}(\mathsf{msks}, \mathsf{tk})$*: the key derivation algorithm generates derives the keys-account data tuple given the master keys set* msks *owning the account and a token* tk *of the account.*
- $(\mathsf{tx}, \mathsf{TK}) \leftarrow \mathsf{TxGen}(\mathsf{st}, P, R, \mathcal{S}, \mathcal{T})$*: the transaction algorithm inputs a state* st*, a predicate* $P : \mathbb{Z}^S \times \mathbb{Z}^T \rightarrow \{0,1\}$*, an index set* R *called the ring, a set of source accounts information* $\mathcal{S} = \{(\mathsf{sks}_i, r_i, \mathsf{accd}_i)\}_{i \in S}$ *and a set of target account information* $\mathcal{T} = \{\mathsf{mpks}_i, \mathsf{accd}'_i\}_{i \in T}$*. Each account is associated with data of the form* $\mathsf{accd} := (a, \mathsf{tout}, b)$*, where* a *denotes the account's asset balance,* $\mathsf{tout} \in \mathbb{N}$ *specifies an epoch-based timeout for ownership logic, and the bit* $b \in \{0,1\}$ *indicates the account type - where 0 corresponds to a standard account, and 1 to a commit-based account. The algorithm outputs a transaction* tx *and the set of tokens* $\mathsf{TK} = \{\mathsf{tx}_i\}_{i \in T}$ *of the target accounts. The transaction* tx *is intended to be published to all users of the system, while the token* tk_i *is meant to be securely exchanged to the owner of* mpk *for* $i \in T$*.*
- $(b) \leftarrow \mathsf{Vf}(\mathsf{st}, \mathsf{tx}, \mathsf{time})$*: The verification algorithm outputs a bit* $b \in \{0,1\}$ *deciding whether the transaction* tx *is valid relative to the state* st *and the given input epoch* $\mathsf{time} \in \mathbb{N}$*.*
- $(b, \mathsf{st}') \leftarrow \mathsf{TimeVf}(\mathsf{st}, \mathsf{tx})$*: The time verification algorithm extracts the current epoch from the given state and inputs it to the* Vf *algorithm, outputing the result bit* $b \in \{0,1\}$ *and a possibly updated state* st'*.*
- $\mathsf{AC}_U \leftarrow \mathsf{StExt}(\mathsf{st})$*: The state extraction algorithm extracts the set of universe accounts* $\mathsf{AC}_U = \{\mathsf{ac}_i\}_{i \in U}$ *encoded in the state* st*.*
- $\mathsf{AC}_T \leftarrow \mathsf{TxExt}(\mathsf{tx})$*: The transaction extraction algorithm extracts the set of target accounts* $\mathsf{AC}_T = \{\mathsf{ac}_i\}_{i \in T}$ *encoded in the state* st*.*
- $\mathsf{time} \leftarrow \mathsf{TimeExt}(\mathsf{tx})$*: The time extraction algorithm extracts the epoch* $\mathsf{time} \in \mathbb{N}$ *encoded in the state* st*.*
- $\mathsf{tout} \leftarrow \mathsf{ExtAccTout}(\mathcal{A})$*: The account timeout extraction algorithm takes as input set of account data* $\mathcal{A} = \{\mathsf{accd}_i\}_{i \in \mathcal{A}}$ *and performs a consistency check to verify that the timeout values* tout *are identical across all commit-based and standard accounts in the set. If this condition holds, the algorithm returns the common timeout epoch* $\mathsf{tout} \in \mathbb{N}$ *associated with the commit-based accounts (if any exist), or returns* 0 *if there are no commit-based accounts. If the timeout values are inconsistent, the algorithm returns* $\perp$*.*
- $e \leftarrow \mathsf{TimedOut}(\mathsf{time}, \mathsf{tout})$*: The timed out algorithm outputs a bit* $e \in \{0,1\}$ *determining whether* tout *is non-zero and* $\mathsf{tout} < \mathsf{time}$*, with* $\mathsf{tout}, \mathsf{time} in \mathbb{N}$*.*
- $\mathcal{Z} \leftarrow \mathsf{EvalTags}(\mathsf{sks})$*: The tag evaluation algorithm outputs the set of tags* $\mathcal{Z} = \{\zeta_i\}_{i \in \mathsf{sks}}$ *evaluated on the given tuple of secret keys* sks*.*

- $b \leftarrow \mathsf{SrcChk}(\mathsf{ac}, \mathsf{sc}, \mathcal{Z}, \mathsf{tout}, e)$*: The source checking takes as input the account* ac*, along with the related keys and account data encaspulated in* $\mathsf{sc} := (\mathsf{sk_s}, r, \mathsf{accd})$*, a set of tags* $\mathcal{Z}$*, an epoch timeout* $\mathsf{tout} \in \mathbb{N}$*, and an expiration bit* $e \in \{0,1\}$*. The algorithm outputs a bit* $b \in \{0,1\}$ *deciding whether to accept or reject that the account* ac *is associated to the provided secret keys* $\mathsf{sk_s}$ *and tags* $\mathcal{Z}$*, given the epoch* tout *and the expiration status* e*, and that the account data* accd *has been commited with randomness* r*.*
- $b \leftarrow \mathsf{TgtChk}(\mathsf{ac}, \mathsf{tk}, \mathsf{accd})$*: The target checking algorithms outputs a bit* $b \in \{0,1\}$ *deciding whether to accept or reject that* accd *has been commited in* ac*.*

5.2 Correctness

Definition 3 (Correctness)**.** *Let* $\mathcal{P}$ *be a family of predicates. A RingCCT scheme* Ω *is* $\mathcal{P}$*-correct if all of the following holds for any* $\lambda \in \mathbb{N}$ *and any* $(pp, *) \in \mathsf{Setup}(1^\lambda)$*.*

Derivation correctness. For any $(\mathsf{mpk}_i, \mathsf{msk}_i) \in \mathsf{KGen}(\mathsf{pp})$ with $i \in \{0,1\}$ let $(\mathsf{sks}, \mathsf{ac}, \mathsf{tk}, \mathsf{accd}, \mathsf{accd}')$ be a tuple satisfying $(\mathsf{sks}, \mathsf{accd}') \in \mathsf{KDer}(\{\mathsf{msk}_i\}, \mathsf{tk})$ and $\mathsf{TgtChk}(\mathsf{ac}, \mathsf{tk}, \mathsf{accd}) = 1$.
Then it follows $\mathsf{accd} = \mathsf{accd}'$ and that for every $\mathsf{pk} \in \mathsf{pks}$ there exists an $\mathsf{sk} \in \mathsf{sks}$ such that $\mathsf{pk} = \Delta.\mathsf{KGen}(\mathsf{pk})$.

TxGeneration correctness. Define the set V_{pp} to be the collection of all tuples (st, P, R, S, T), satisfying the following properties:

- $\mathsf{P} \in \mathcal{P}$
- $\mathsf{P}(\mathsf{a}_S, \mathsf{a}'_T) = 1$
- $S \subseteq R \subseteq U$
- $\mathsf{SrcChk}(\mathsf{StExt}(\mathsf{st})[i], (\mathsf{sks}_i, r_i, \mathsf{accd}_i), \mathsf{EvalTags}(\mathsf{sks}_i), \mathsf{tout}, e)$ for all $i \in S$.

where $\mathcal{S} = \{\mathsf{sks}_i, \mathsf{accd}_i\}$, $\mathcal{T} = \{\mathsf{mpks}'_i, \mathsf{accd}'_i\}$, $\mathsf{tout} = \mathsf{ExtAccTout}(\{\mathsf{accd}_i\}_{i \in S})$ and $e = \mathsf{TimedOut}(\mathsf{TimeExt}(\mathsf{st}), \mathsf{tout})$.
Then for any $(\mathsf{st}, P, R, \mathcal{S}, \mathcal{T}) \in V_{\mathsf{pp}}$, and any $(\mathsf{tx}, \mathsf{tk}) \in \mathsf{TxGen}(\mathsf{st}, P, R, \mathcal{S}, \mathcal{T})$, if $(b, \mathsf{st}') = \mathsf{TimeVf}(\mathsf{st}, \mathsf{tx})$, the following holds:

- $b = 1$
- $\mathsf{TxExt}(\mathsf{tx}) \subseteq \mathsf{StExt}(\mathsf{st}')$
- $\mathsf{TgtChk}(\mathsf{TxExt}(\mathsf{tx}[i], \mathsf{tk}_i, \mathsf{accd}_i) = 1$ for all i $\in T$

5.3 Security

We here define the security properties of RingCCT.

Balance. Balance ensures correct account ownership and prevents of double-spending and overspeding. Specifically, an account may only spend assets that it owns and have not been previously spent. We first require that the source checking algorithms SrcChk is computationally binding to a set of secret keys and an associated amount, while the target checking algorithm TgtChk is binding solely to an amount. We then formalise the balance property via the security experiment $\mathsf{Balance}_{\Omega,\mathcal{P},\mathcal{A},\mathcal{E}_\mathcal{A}}$, where $\mathcal{A}$ denotes an adversary and $\mathcal{E}_\mathcal{A}$ a knowledge extractor. The adversary $\mathcal{A}$ generates a sequence of valid transactions tx_i for $i \in \mathbb{Z}_l$. The knowledge extractor $\mathcal{E}_\mathcal{A}$ subsequently extracts the information associated with the source and target accounts for every transaction. The experiments returns 1 if some source or target account is ill-formed or if there exist distinct $i < i'$ such that the sets of source accounts used in the transactions tx_i and tx_i overlap, which indicates a double-spending event.

Definition 4 (Balance)**.** *A RingCCT scheme is balanced if:*
1. The source checking algorithm SrcChk *computationally binds an account to the stored amount and a set*

of secret keys determined by the account type b and epoch timeout tout*, that is for any PPT adversary* $\mathcal{A}$ *it holds that*

$$\mathsf{Pr}\left[\begin{array}{l|r} \begin{cases} \mathsf{SrcChk}(\mathsf{ac},\mathsf{sc},\mathcal{Z}_S,\mathsf{tout},e)=1 \\ \mathsf{SrcChk}(\mathsf{ac},\mathsf{sc}',\mathcal{Z}'_S,\mathsf{tout},e)=1 \\ \mathsf{accd}\neq\mathsf{accd}' \ \vee \mathsf{ssk}\neq\mathsf{ssk}' \ \vee (b=1\wedge \\ (e=1\wedge\mathsf{rsk}\neq\mathsf{rsk}') \ \vee (e=0\wedge\mathsf{tsk}\neq\mathsf{tsk}')) \end{cases} & \begin{array}{r} (\mathsf{pp},\mathsf{st})\leftarrow\mathsf{Setup}(1^\lambda) \\ (\mathsf{ac},\mathsf{sc},\mathsf{sc}',\mathcal{Z}_S,\mathcal{Z}'_S,\mathsf{tout},e)\leftarrow\mathcal{A}(\mathsf{pp}) \end{array} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{sc}=(\mathsf{sks},r,\mathsf{accd})$ *and* $\mathsf{sks}=(\mathsf{ssk},\mathsf{rsk},\mathsf{tsk})$.

2. The target checking algorithm TgtChk *computationally binds an account to the stored amount and account data, that is for any PPT adversary* $\mathcal{A}$ *it holds that*

$$\mathsf{Pr}\left[\begin{array}{l|r} \begin{cases} \mathsf{TgtChk}(\mathsf{ac},\mathsf{tk},(a,\mathsf{tout},b))=1 \\ \mathsf{TgtChk}(\mathsf{ac},\mathsf{tk}',(a',\mathsf{tout}',b'))=1 \\ (a,\mathsf{tout},b)\neq(a',\mathsf{tout}',b') \end{cases} & \begin{array}{r} (\mathsf{pp},\mathsf{st})\leftarrow\mathsf{Setup}(1^\lambda) \\ (\mathsf{ac},\mathsf{tk},\mathsf{tout},a,b)\leftarrow\mathcal{A}(\mathsf{pp}) \\ (\mathsf{ac},\mathsf{tk}',\mathsf{tout}',a',b')\leftarrow\mathcal{A}(\mathsf{pp}) \end{array} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

3. For any PPT adversary $\mathcal{A}$ *there exists an expected polynomial-time extractor such that*

$$\mathsf{Pr}\left[\mathsf{Balance}_{\Omega,\mathcal{P},\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda)=1\right]\leq\mathsf{negl}(\lambda)$$

where $\mathsf{Pr}\left[\mathsf{Balance}_{\Omega,\mathcal{P},\mathcal{A},\mathcal{E}_\mathcal{A}}\right]$ *is defined in* **??**.

The transaction verification is performed by extracting the current epoch from the state via the time extraction algorithm $\mathsf{TimeExt}$ and then by running the verification algorithm TimeVf. This implies that the balance security properties covers both standard and commit accounts.

$\mathsf{Balance}_{\Omega,\mathcal{P},\mathcal{A},\mathcal{E}_\mathcal{A}}$

$(\mathsf{pp}, \mathsf{st}_0) \leftarrow \mathsf{Setup}(1^\lambda)$

$(\mathsf{tx}_i)_{i\in\mathbb{Z}_l} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{st}_0)$

$(P_i, R_i, S_i, T_i)_{i\in\mathbb{Z}_l} \leftarrow \mathcal{E}_\mathsf{A}(\mathsf{pp}, \mathsf{st}_0, (\mathsf{tx}_i)_{i\in\mathbb{Z}_l})$

$\{\mathsf{sc}_{i,j} := (\mathsf{sks}_{i,j}, r_{i,j}, \mathsf{accd}_{i,j})\}_{j\in S_i} := \textbf{parse}\ (S_i)_{i\in\mathbb{Z}_l}$

$\{\mathsf{mpks}_{i,j}, \mathsf{tk}_{i,j}, \mathsf{accd}'_{i,j}\}_{j\in T_i} := \textbf{parse}\ (T_i)_{i\in\mathbb{Z}_l}$

for $t \in \mathbb{Z}_l$ **do** $(b_t, \mathsf{st}_{t+1}) := \mathsf{TimeVf}(\mathsf{st}_\mathsf{t}, \mathsf{tx}_\mathsf{t})$

for $i \in \mathbb{Z}_l$ **do**

$\mathsf{accd}_{i,S_i} := (\mathsf{accd}_{i,j})_{j\in S_i}, \mathsf{tout}_i := \mathsf{ExtAccTout}(\{\mathsf{accd}_{i,j}\}_{j\in S})$

$e_i := \mathsf{TimedOut}(\mathsf{TimeExt}(\mathsf{st}_i), \mathsf{tout}), \mathsf{accd}'_{i,T_i} := (\mathsf{accd}'_{i,j})_{j\in T_i}$

$\{\mathsf{ac}_{i,j}\}_{j\in U_i} := \mathsf{StExt}(\mathsf{st}_i), \{\mathsf{ac}'_{i,j}\}_{j\in T_i} := \mathsf{TxExt}(\mathsf{tx}_i)$

$$b'_i := \begin{cases} \mathsf{TxExt} \subseteq \mathsf{StExt}(\mathsf{st}_{i+1}) \\ P_i \in \mathcal{P} \\ P_i(\mathsf{accd}_{i,S_i}, \mathsf{accd}'_{i,T_i}) \\ S_i \subseteq R_i \subseteq U_i \\ \mathsf{SrcChk}(\mathsf{StExt}(\mathsf{st}_i)[j], \mathsf{sc}_{i,j}, \mathsf{EvalTags}(\mathsf{sks}_{i,j}), \mathsf{tout}_i, e_i) = 1 \quad \forall j \in \mathsf{S}_i \\ \mathsf{TgtChk}(\mathsf{TxExt}(\mathsf{tx}_i)[j], \mathsf{tk}_{i,j}, \mathsf{accd}_{i,j}) = 1 \quad \forall j \in \mathsf{T}_i \end{cases}$$

$b'' := (\exists i_0 < i_1, S_{i_0} \cap S_{i_1} = \emptyset)$

return $\bigwedge_{i\in\mathbb{Z}_l} b_i \wedge \neg(\bigwedge_{i\in\mathbb{Z}_l} b'_i \wedge b''_i)$

Figure 3: Balance experiment definition

Privacy. Privacy captures spender and receiver anonymity as well as assets confidentiality. The property is modeled by the security experiment $\mathsf{Privacy}^b_{\Omega,\mathcal{A}}$, which is parameterised by the bit b. The experiments begins by initialising the RingCCT system state st through the Setup algorithm. The adversary is then gives access to oracles for account generation, corruption, transaction and verification, as defined in **??**.
Commit and standard accounts can be generated by calling $\mathsf{CommAccGen}\mathcal{O}$ and $\mathsf{AccGen}\mathcal{O}$ respectively, both of which return the set of associated public keys. Existing accounts can be corrupted via $\mathsf{Corr}\mathcal{O}$, which returns the corresponding secret keys unless the account belongs to the set ID^*. The verification oracle $\mathsf{TimeVf}\mathcal{O}$ takes a transaction tx as input and updates the system state st if the given transaction is sucessfully verified with the verification algorithm TimeVf.
The transaction oracle $\mathsf{TxGen}\mathcal{O}$ takes as input the tuple $(P, R, \mathcal{S}', \mathcal{S}^*, \mathcal{T}', \mathcal{T}^*)$ provided $\mathcal{A}$, where $\mathcal{S}'$ and $\mathcal{T}'$ are sets of source and target accounts with keypairs generated by the adversary, and $\mathcal{S}^*$, $\mathcal{T}^*$ instructs the oracle to retrieve keypairs of uncorrupted accounts. The oracle combines the given sets, generates the transaction and returns it alongside the tokens associated with the target accounts.
The adversary then generates a pair of transactions with inputs $(P, R, \mathcal{S}'_i, \mathcal{S}^*, \mathcal{T}'_i, \mathcal{T}^*)$, where $i \in \{0, 1\}$, differing only in the selected set of honest source and target accounts. Honest accounts that are involved in verified transactions are recorded and blocked by including them in the set AC^*.
The bit-parametised transaction is then returned to the adversary $\mathcal{A}$, who may further interact with oracles before outputting a bit that represents the outcome of the experiment.

Definition 5 (Privacy)**.** *A RingCCT scheme is private if for all PPT adversaries $\mathcal{A}$ it holds that*

$$\mid \Pr\left[\mathsf{Privacy}^0_{\Omega,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Privacy}^1_{\Omega,\mathcal{A}}(1^\lambda) = 1\right] \mid \leq \mathsf{negl}(\lambda)$$

Where $\mathsf{Privacy}^b_{\Omega,\mathcal{A}}$ is defined in **??**.

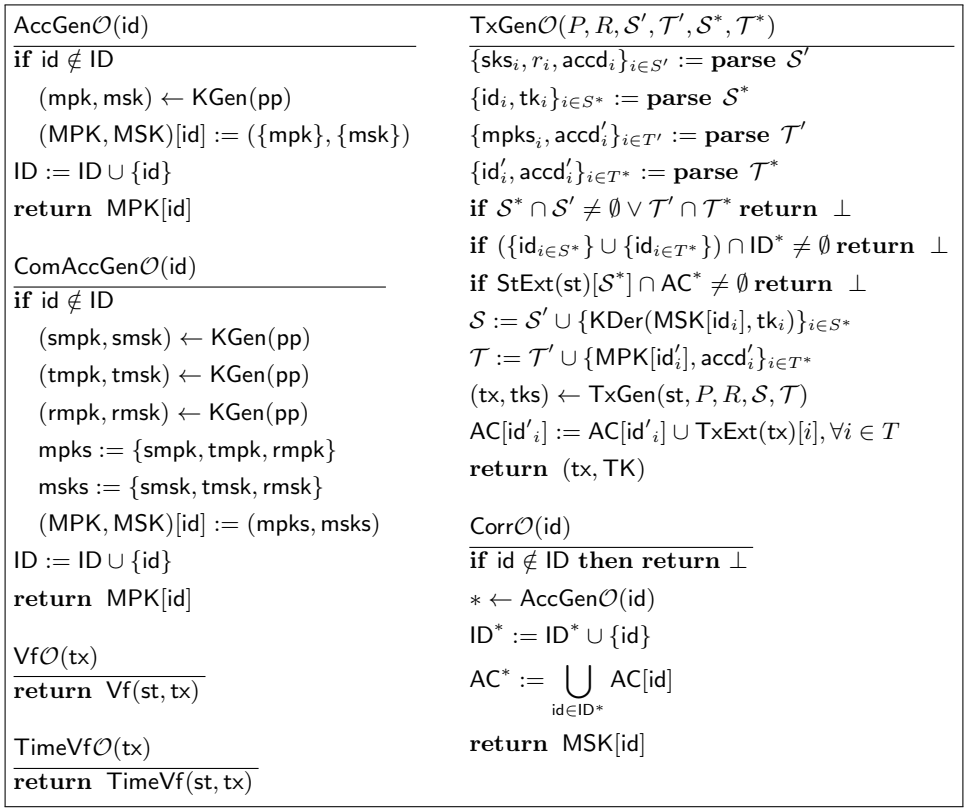

Figure 4: Oracles for the privacy and availability experiments.

$\mathsf{Privacy}^{b}_{\mathcal{A}}$

$(\mathsf{pp}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda)$

$\mathcal{O} := \{\mathsf{AccGen}\mathcal{O}, \mathsf{CommAccGen}\mathcal{O}, \mathsf{Corr}\mathcal{O}, \mathsf{TxGen}\mathcal{O}, \mathsf{TimeVf}\mathcal{O}\}$

$(P, R, \mathcal{S}', \mathcal{T}', (\mathcal{S}^*_i, \mathcal{T}^*_i)_{i\in\{0,1\}}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pp})$

for $i \in \{0, 1\}$

  $(\mathsf{tx}_i, *) \leftarrow \mathsf{TxGen}\mathcal{O}(P, R, \mathcal{S}', \mathcal{T}', \mathcal{S}^*_i, \mathcal{T}^*_i)$

  $(b_i, \mathsf{st}'_i) := \mathsf{TimeVf}(\mathsf{st}, \mathsf{tx}_i)$

  if $b_i = 0$ **return** 0

$\{\mathsf{id}_{i,j}, \mathsf{tk}_{i,j}\}_{j\in S^*_i} := \textbf{parse } \mathcal{S}^*_i$

$\{\mathsf{id}'_{i,j}, \mathsf{accd}_{i,j}\}_{j\in S^*_i} := \textbf{parse } \mathcal{T}^*_i$

$\mathsf{ID}^* := \mathsf{ID}^* \cup \{\mathsf{id}_{i,j}\}_{j\in S^*_i} \cup \{\mathsf{id};_{i,j}\}_{j\in T^*_i}$

$\mathsf{AC}^* := \mathsf{AC}^* \cup \mathsf{StExt}(\mathsf{st})[\mathcal{S}^*_i] \cup \mathsf{TxExt}(\mathsf{tx}_i)[\mathcal{T}^*_i]$

if $(|\mathcal{S}^*_i| \neq |\mathcal{S}^*_\infty|) \vee (|\mathsf{StExt}(\mathsf{st}'_0)\ \mathsf{StExt}(\mathsf{st})| \neq |\mathsf{StExt}(\mathsf{st}'_1)\ \mathsf{StExt}(\mathsf{st})|)$ **return** 0

$b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{tx}_b)$

return b'

Figure 5: Privacy experiment with oracles defined in **??**.

Definition 6 (Availability)**.** *A RingCCT scheme is available if for all PPT adversaries $\mathcal{A}$ it holds that*

$$\Pr\left[\mathsf{Availability}_{\Omega,\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

Where $\mathsf{Availability}_{\Omega,\mathcal{A}}$ *is defined in* **??**.

$\mathsf{Availability}_{\Omega,\mathcal{A}}$

$(\mathsf{pp}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda)$

$\mathcal{O} := \{\mathsf{KGen}\mathcal{O}, \mathsf{Corr}\mathcal{O}, \mathsf{TxGen}\mathcal{O}, \mathsf{TimeVf}\mathcal{O}\}$

$(P, R, \mathcal{S}, \mathcal{T}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pp})$

$(\mathsf{tx}, \mathsf{TK}) \leftarrow \mathsf{TxGen}\mathcal{O}(P, R, \mathcal{S}, \mathcal{T})$

$\{\mathsf{id}_j, \mathsf{tk}_j\}_{j\in S} := \textbf{parse } \mathcal{S}$

if $\mathcal{S}^* \not\subseteq U$ **return** 0

$(\mathsf{ID}^*, \mathsf{AC}^*) := (\{\mathsf{id}_j\}_{j\in S}, \mathsf{StExt}(\mathsf{st})[\mathcal{S}])$

$\mathsf{time} := \mathsf{TimeExt}(\mathsf{st})$

$b := \mathsf{Vf}(\mathsf{st}, \mathsf{tx}, \mathsf{time})$

$\perp \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{tx}, \mathsf{TK})$

$b' := \mathsf{Vf}(\mathsf{st}, \mathsf{tx}, \mathsf{time})$

return $b \wedge \neg b'$

Figure 6: Availability experiment with oracles defined in **??**.

Availability. Availability captures the notion that a valid transaction created at a given point in time should remain valid in the future, unless the associated output has been spent.This provides security against *denial-of-spending* attacks, wherein an adversary attempts to prevent a specific user or account from spending their assets. We note that in the RingCCT schemes incorporates two verification algorithms, TimeVf

and Vf. The former verifies against the current epoch of the given state, while the latter takes an input an arbitrary epoch. Given that commit-type accounts inherently violate future availability by design, we only require that the validity of a transaction with respect to some epoch is valid at any point in time. Consequently, the experiment $\mathsf{Availability}_{\Omega,\mathcal{A}}$ is defined with respect to Vf

The security experiment $\mathsf{Availability}_{\Omega,\mathcal{A}}$ is described in **??**. Similarly to the privacy experiments, we initialise the RingCCT system and provide the adversary $\mathcal{A}$ access to the same set of oracles. Following oracle interaction, the adversary $\mathcal{A}$ specifies the input for the oracle $\mathsf{TxGen}\mathcal{O}$, which computes tx. The set of source accounts used by the $\mathcal{A}$ must be honest and are subsequently blocked once included in the challenge transaction tx. The adversary $\mathcal{A}$ receive the transaction tx and further interacts with the oracles, which may update the system state. After $\mathcal{A}$ halts, the transaction tx is verified against the previous state's time with the result denoted by b'. The experiment returns $b' \wedge b$, capturing whether the transaction is valid for some state st but rejected for st'.

5.4 Construction

In this section we construct a RingCCT scheme Ω from a tagging scheme Δ, a commitment scheme Γ and a non-interactive argument system Π. The global system state st consists of the set of all universe accounts AC_U, the set $\mathcal{Z}_U$ containing all valid tags generated by Δ in previously accepted transactions, and the variable $\mathsf{time} \in \mathbb{N}$ representing the epoch of the system.
Each spent account in the system will be associated with one or more tags recorded in $\mathcal{Z}_U$, with each tag computationally binding to the account's corresponding secret key. The master public and secret keys of the RingCCT scheme are public and secret keys of Δ respectively.
An account ac is characterized by the tuple of public keys $\mathsf{pks} := (\mathsf{spk}, \mathsf{tpk}, \mathsf{rpk})$ and a commitment $\mathsf{co} := \mathsf{Com}(\mathsf{accd}, r)$, where $\mathsf{accd} := (a, \mathsf{tout}, b)$ represents the commited account data. Here, a denotes the asset amount owned by the account, tout defines a timeout timeout, and b is a bit specifying the type of the account (i.e., standard or commit-type). The account is associated to the corresponding tuple of secret keys $\mathsf{sks} := (\mathsf{ssk}, \mathsf{tsk}, \mathsf{rsk})$, where for each secret key in the tuple sks we have the corresponding public key in pks computed with $\Delta.\mathsf{Eval}$. Each of the secret key in sks the account data are derivable from the master secret key msk and the token $\mathsf{tk} := (r, \delta, \mathsf{accd})$ by computing $\mathsf{sk} := \mathsf{msk} + \delta$
Fix any predicate family $\mathcal{P}$ and let $P \in \mathcal{P}$. To create a transaction, the TxGen algorithm computs the tags of the corresponding secret keys $\mathsf{sk}_i + \delta_i$ for all source accounts $i \in S$ and creates the target accounts ac. It then generates a non-interactive argument of following relation:

$$\mathcal{R}(\mathsf{stmnt}, \mathsf{wit}) := \begin{cases} S \subseteq R \\ \mathsf{SrcChk}(\mathsf{ac}_i, \mathsf{sc}_i, \mathcal{Z}_i, \mathsf{tout}, e) = 1 \qquad \forall i \in S \\ \mathsf{TgtChk}(\mathsf{ac}'_i, \mathsf{tk}_i, \mathsf{accd}'_i) = 1 \qquad \forall i \in T \\ P(a_S, a'_T) = 1 \end{cases}$$

where the statement stmnt and witness wit are of the form

$$\mathsf{stmnt} := (P, \mathsf{AC}_R, \mathcal{Z}_{\bar{S}}, \mathsf{AC}_T, \mathsf{tout}, e)$$

$$\mathsf{wit} := ((\mathsf{sc}_i)_{i \in S}, (\mathsf{tk}_i, \mathsf{accd}'_i)_{i \in T})$$

respectively, where $\mathsf{sc}_i = (\mathsf{sks}_i, r_i, \mathsf{accd}_i)$. To verify a transaction, the verifier checks that the predicate P is admissible, the ring R is a subset of the universe accounts $R \subseteq U$, the proof for the above relation is sound, and that no tags ζ given in the transaction appeared before, i.e. $\mathcal{Z}_S \cup \mathcal{Z}_U = \emptyset$. If these checks are passed, the verified would agree to advance the state from st to $\mathsf{st}' := (\mathsf{AC}_T \cup \mathsf{AC}_T ...)$

rlai: incomplete

$\mathsf{Setup}(1^\lambda)$

$\mathsf{crs} \leftarrow \Pi.\mathsf{Setup}(1^\lambda)$

$\mathsf{ck} \leftarrow \Gamma.\mathsf{Gen}(1^\lambda)$

$\mathsf{pp}_\Delta \leftarrow \Delta.\mathsf{Setup}(1^\lambda)$

return $(\mathsf{pp}, \mathsf{st})$

$\mathsf{TimeExt}(\mathsf{st})$

$(\mathsf{AC}_U, \mathcal{Z}_U, \mathsf{time}) := \textbf{parse}\ \mathsf{st}$

return time

$\mathsf{StExt}(\mathsf{st})$

$(\mathsf{AC}_U, \mathcal{Z}_U, \mathsf{time}) := \textbf{parse}\ \mathsf{st}$

return AC_U

$\mathsf{TimedOut}(\mathsf{tout}, \mathsf{time})$

return $\mathsf{tout} \neq 0 \wedge \mathsf{tout} \overset{?}{<} \mathsf{time}$

$\mathsf{KGen}(\mathsf{pp})$

$\mathsf{msk} \leftarrow\!\!\$\ \mathcal{K}$

$\mathsf{mpk} := \Delta.\mathsf{KGen}(\mathsf{msk})$

return $(\mathsf{mpk}, \mathsf{msk})$

$\mathsf{TxExt}(\mathsf{tx})$

$(P, R, \mathsf{AC}_T, \mathcal{Z}_{\bar{S}}) := \textbf{parse}\ \mathsf{tx}$

return AC_T

$\mathsf{ExtAccTout}(\mathcal{A})$

$\{(a_i, \mathsf{tout}_i, b_i)\}_{i\in\mathcal{A}} := \textbf{parse}\ \mathcal{A}$

assert $\nexists\, i, j \in \mathcal{A} \mid (\mathsf{tout}_i, b_i) \neq (\mathsf{tout}_j, b_j)$

$\mathsf{tout} := a \in \{\mathsf{tout}_i\}_{i\in\mathcal{A}} \mid a \neq 0$

if $\mathsf{tout} = \bot$

  return 0

return tout

$\mathsf{EvalTags}(\mathsf{sks})$

return $\{\Delta.\mathsf{Eval}(\mathsf{sk}_i)\}_{i\in\mathsf{sks}}$

6 Instantiation and Performance Evaluation

6.1 Instantiation

Commitment

Tagging Scheme

$P_0(\mathsf{ssk}^0_{\mathbb{A}}, \mathsf{spk}^1_{\mathbb{B}}, \mathsf{tsk}_{\mathbb{B}})$ $\qquad$ $P_1(\mathsf{ssk}^0_{\mathbb{B}}, \mathsf{spk}^1_{\mathbb{A}}, \mathsf{tx})$

Figure 7: Protocol definition of 2PC (RingCCT - public) $\Gamma_{\mathsf{CommitTx}}$

$P_0(\mathsf{ssk}^0_{\mathbb{A}}, \mathsf{rsk}_{\mathbb{A}}, \mathsf{tsk}_{\mathbb{B}})$ $\qquad$ $P_1(\mathsf{ssk}_{\mathbb{B}}, \mathsf{rsk}_{\mathbb{B}}, \mathsf{tsk}_{\mathbb{A}})$

Figure 8: Protocol definition of 2PC (RingCCT - RingCCT) $\Gamma_{\mathsf{CommitTx}}$

2PC

> Write down explicitly (i.e. in terms of $\mathbb{G}$ and $\mathbb{Z}_q$ arithmetic) for which functionalities do we need 2PCs.

Zero-Knowledge Proofs

$\mathsf{Vf}(\mathsf{st}, \mathsf{tx}, \mathsf{time})$

$(\mathsf{AC}_U, \mathcal{Z}_U, \bot) := \textbf{parse}\ \mathsf{st}$
$\{\mathsf{ac}_i\}_{i\in U} := \textbf{parse}\ \mathsf{AC}_U$
$(P, R, \mathsf{AC}_T, \mathcal{Z}_{\bar{S}}, \mathsf{tout}, \pi) := \textbf{parse}\ \mathsf{tx}$
$\mathsf{AC}_R := \{\mathsf{ac}_i\}_{i\in R}$
$e := \mathsf{TimedOut}(\mathsf{tout}, \mathsf{time})$
$\mathsf{stmnt} := (P, \mathsf{AC}_R, \mathsf{AC}_T, \mathcal{Z}_{\bar{S}}, \mathsf{tout}, e)$
$$\textbf{if}\ \begin{cases} P \in \mathcal{P} \\ R \subseteq U \\ \Pi.\mathsf{Vf}(\mathsf{crs}, \mathsf{stmnt}, \pi) = 1 \\ \mathcal{Z}_{\bar{S}} \cap \mathcal{Z}_{\bar{U}} = \emptyset \end{cases} \textbf{then}$$
$\quad \textbf{return}\ (1, \mathsf{st}')$
$\textbf{else return}\ (0, \mathsf{st})$

$\mathsf{KDer}(\mathsf{msks}, \tau)$

$(r, \delta, \mathsf{accd}) := \textbf{parse}\ \tau$
$\mathsf{sks} := \{\mathsf{msk}_i + \delta\}_{i\in\mathsf{msks}}$
$\textbf{return}\ (\mathsf{sks}, r, \mathsf{accd})$

$\mathsf{TgtChk}(\mathsf{ac}, \mathsf{tk}, \mathsf{accd})$

$(\mathsf{pks}, \mathsf{co}) := \textbf{parse}\ \mathsf{ac}$
$(r, \delta, \mathsf{accd}') := \textbf{parse}\ \mathsf{tk}$
$$\textbf{return}\ \begin{cases} \mathsf{accd}' \stackrel{?}{=} \mathsf{accd} \\ \mathsf{co} \stackrel{?}{=} \Gamma.\mathsf{Com}(\mathsf{accd}, r) \end{cases}$$

$\mathsf{SrcChk}(\mathsf{ac}, \mathsf{sc}, \mathcal{Z}, \mathsf{tout}, e)$

$(\mathsf{sks}, r, \mathsf{accd}) := \textbf{parse}\ \mathsf{sc}$
$(a, \mathsf{tout}', b) := \textbf{parse}\ \mathsf{accd}$
$(\mathsf{pks}, \mathsf{co}) := \textbf{parse}\ \mathsf{ac}$
$(\mathsf{ssk}, \mathsf{tsk}, \mathsf{rsk}) := \textbf{parse}\ \mathsf{sks}$
$(\mathsf{spk}, \mathsf{tpk}, \mathsf{rpk}) := \textbf{parse}\ \mathsf{pks}$
$\textbf{assert}\ \mathsf{co} = \Gamma.\mathsf{Com}(\mathsf{accd}, r)$
$\textbf{assert}\ b = 0 \vee (e \neq 0 \wedge \mathsf{tout}' = \mathsf{tout})$
$\textbf{if}\ b = 0$
$$\quad \textbf{return}\ \begin{cases} \mathsf{spk} \stackrel{?}{=} \Delta.\mathsf{KGen}(\mathsf{ssk}) \\ \Delta.\mathsf{Eval}(\mathsf{ssk}) \stackrel{?}{\in} \mathcal{Z} \end{cases}$$
$\textbf{else if}\ e = 1$
$$\quad \textbf{return}\ \begin{cases} \mathsf{tpk} \stackrel{?}{=} \Delta.\mathsf{KGen}(\mathsf{tsk}) \\ \mathsf{spk} \stackrel{?}{=} \Delta.\mathsf{KGen}(\mathsf{ssk}) \\ \Delta.\mathsf{Eval}(\mathsf{ssk}), \Delta.\mathsf{Eval}(\mathsf{tsk}) \stackrel{?}{\in} \mathcal{Z} \end{cases}$$
$\textbf{else}$
$$\quad \textbf{return}\ \begin{cases} \mathsf{rpk} \stackrel{?}{=} \Delta.\mathsf{KGen}(\mathsf{rsk}) \\ \Delta.\mathsf{Eval}(\mathsf{ssk}), \Delta.\mathsf{Eval}(\mathsf{rsk}) \stackrel{?}{\in} \mathcal{Z} \end{cases}$$

$\mathsf{TimeVf}(\mathsf{st}, \mathsf{tx})$

$\mathsf{time} := \textbf{parse}\ \mathsf{TimeExt}$
$\textbf{return}\ \mathsf{Vf}(\mathsf{st}, \mathsf{tx}, \mathsf{time})$

$\mathsf{TxGen}(\mathsf{st}, P, R, \mathcal{S}, \mathcal{T})$

$\{\mathsf{sc}_i := (\mathsf{sks}_i, r_i, \mathsf{accd}_i)\}_{i\in S} := \textbf{parse}\ \mathcal{S}$
$\{(\mathsf{mpks}_i, \mathsf{accd}'_i)\}_{i\in T} := \textbf{parse}\ \mathcal{T}$
$\{\mathsf{ac}_i\}_{i\in U} := \mathsf{StExt}(\mathsf{st})$
$\mathsf{time} := \mathsf{TimeExt}(\mathsf{st})$
$\mathsf{tout} \leftarrow \mathsf{ExtAccTout}(\{\mathsf{accd}_i\}_{i\in S})$
$\textbf{assert}\ \mathsf{tout} \neq \bot$
$e \leftarrow \mathsf{TimedOut}(\mathsf{tout}, \mathsf{time})$
$\textbf{for}\ i \in T\ \textbf{do}$
$\quad r'_i \leftarrow\!\!\$\ \chi$
$\quad \delta_i \leftarrow\!\!\$\ \mathcal{K}$
$\quad \mathsf{co}_i := \Gamma.\mathsf{Com}(\mathsf{accd}'_i, r'_i)$
$\quad (a_i, \mathsf{tout}, b_i) := \textbf{parse}\ \mathsf{accd}'_i$
$\quad (\mathsf{smpk}_i, \mathsf{tmpk}_i, \mathsf{rmpk}_i) := \textbf{parse}\ \mathsf{mpks}_i$
$\quad \mathsf{spk}_i := \mathsf{smpk}_i + \Delta.\mathsf{Eval}(\delta_i)$
$\quad \mathsf{tk}_i := (r'_i, \delta_i, \mathsf{accd}'_i)$
$\quad \textbf{if}\ b \neq 0$
$\quad\quad \mathsf{tpk}_i := \mathsf{tmpk}_i + \Delta.\mathsf{Eval}(\delta_i)$
$\quad\quad \mathsf{rpk}_i := \mathsf{rmpk}_i + \Delta.\mathsf{Eval}(\delta_i)$
$\quad\quad \mathsf{pks}_i := (\mathsf{spk}_i, \mathsf{tpk}_i, \mathsf{rpk}_i)$
$\quad \textbf{else}$
$\quad\quad \mathsf{pks}_i := (\mathsf{spk}_i, \bot, \bot)$
$\quad \mathsf{ac}'_i := (\mathsf{pks}_i, \mathsf{co}'_i)$
$\mathsf{AC}_R := \{\mathsf{ac}_i\}_{i\in R}$
$\mathsf{AC}_T := \{\mathsf{ac}'_i\}_{i\in T}$
$\mathcal{Z}_S := \{\mathsf{EvalTags}(\mathsf{sks}_i)\}_{i\in S}$
$\mathsf{stmnt} := (P, \mathsf{AC}_R, \mathsf{AC}_T, \mathcal{Z}_S, \mathsf{tout}, e)$
$\mathsf{wit} := ((\mathsf{sc}_i)_{i\in S}), (\mathsf{tk}_i, \mathsf{accd}'_i)_{i\in T})$
$\pi \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, \mathsf{stmnt}, \mathsf{wit})$
$\mathsf{tx} := (P, \mathsf{AC}_R, \mathsf{AC}_T, \mathcal{Z}_S, \mathsf{tout}, \pi)$
$\mathsf{TK} := \{\mathsf{tk}_i\}_{i\in T}$
$\textbf{return}\ (\mathsf{tx}, \mathsf{TK})$

6.2 Performance Evaluation