Account-based anonymous rollup with unlinkable transactions

Olivier Bégassat Alexandre Belling Nicolas Liochon

April 2020

1 Introduction

This document describes a zero-knowledge rollup [1, 2] scheme to perform anonymous transactions on Ethereum. This protocol was first presented in [4]. It uses an account-based data structure and a mechanism of "money orders" to achieve transaction unlinkability.

Rollups involve three types of actors: (1) *Operators* manage the rollup state, execute rollup transactions and generate proofs of correct state-root-hash updates. (2) *Users* are account owners in the rollup. They send funds to, and receive funds from, other users. They are not tied to a specific operator. We also include (3) a *blockchain* which executes the layer 1 blockchain consensus, verifies operator produced proofs and houses the rollup smart-contract. Lastly, *Observers* (which include operators, users and outsiders) collect and analyze available information about the rollup state and transactions.

In this protocol, *users* keep private their account data. *Operators* are only aware of account hashes. To transfer, a *sender* creates a *money order*, updates its account locally, *registers* the changes to an operator. Once the update has been executed on the blockchain, the sender sends a *receipt* with a proof of inclusion to the *receiver*. The receiver can later *redeem* the money order, by updating its private account data and sending the update to an operator. The rollup state includes all previous *historical states* in a Merkle tree, thereby allowing users to generate inclusion proofs without leaking the date of money order creation. User can batch money order creations and redemptions, allowing them to hide the real volume of their activity. Moreover this mechanism minimizes Merkle-tree depth.

2 Notations

Let S denote a sender and R denote a receiver. When appropriate, we append "R" or "S" to variable names, thereby indicating that they relate to sender or receiver data. For instance, $\mathsf{account}\mathsf{S}_{\mathsf{hash}}$ denotes the account hash of the sender while $\mathsf{account}\mathsf{R}_{\mathsf{hash}}$ denotes the receiver's account.

- $\mathsf{account}_{\mathsf{hash}}$ is the account hash of a user.
- $\mathsf{account}_{\mathsf{id}}$ is the account id of a user. It corresponds to a position in an account Merkle-tree.
- $\mathsf{balance}$ is the balance of a user.
- pubkey denotes a public key.
- X denotes an amount of tokens transferred.
- $\mathsf{State}_0, \mathsf{State}_1, \mathsf{State}_a, \mathsf{State}_b, \ldots$ denote the state of the rollup at some point in time.
- MO denotes a money order,
- MT denotes a Merkle tree.

Whenever a value v (such as an account hash, a balance, ...) is updated, let v' denote the updated value.

3 Design Goals

3.1 Anonymity

Account data. All account data must be private, that is: known only to the account owner. The operator, in particular, should be able to fulfil its function knowing nothing about the account data. Observers may, however, glean information about account activity. For instance, the issuance of one or more money orders (in a single account operation) modifies the account hash. So does redeeming a one or more money orders (in a single account operation). Thus, an observer will know that an account's hash was updated u times in an interval of time. Furthermore, if m of these modifications are associated with money orders issuance operations and r with money order redemption operations (so that $u = m + r$), an observer will know that that account created $\geq m$ money-orders and redeemed $\geq r$ money orders. The details of these account operations (transfered amounts, recipient of money order, originator of redeemed money-order, ...), however, should not be deducible from updates of the account hash.

3.1.1 Participants in a transaction are unlinkable to an observer

- The receiver of a transfer is properly hidden from any observer.
- The sender of a transfer is hidden from any observer.
- The sender's and the receiver's action cannot be linked together by any observer.

We slightly expand on these points as follows: when an account (a sender) generates a money order creation transaction (which may may insert *several* new money orders into their account and thus into the state), the number of thereby created money orders as well as the accounts they are destined to, can't be deduced by observers. This includes operators and the recipients. Furthermore, that money order creation transaction can't be linked to any *future* money order redemption transactions made by other accounts. That is, when one or more money orders are redeemed, one can't know which historical transactions created these money orders.

3.1.2 Transaction detail privacy

- Sender's actions reveal nothing about the transfer to any observer.
- Receiver's actions reveal nothing about the transfer to any observer.

3.1.3 Data privacy

- An observer with access to all public data cannot learn any information about the contents of an account.

3.2 Safety requirement

- Users cannot register transfers of amounts in excess of their account balance.
- No tokens are created during transfers: if the sender spends X, the receiver receives X.
- Money orders can only be redeemed once.

4 Construction

The protocol has a base layer and can be extended with optional features. In this section we provide an overview on the base layer protocol, i.e. the protocol for unlinkable token transfers. Later we briefly describe optional features.

4.1 Protocol overview

An unlikable transfer of X tokens from a sender to a receiver happens in six steps.

4.1.1 Step 1: money order creation

The sender:

- decreases its account balance by X, inserts one or more money orders for a total token value of X in its money order set and computes the new account root hash,
- updates its randomness,
- generates an **account-hash-update-proof** validating the transition from its current account hash to the new one,
- and sends an **account-hash-update-transaction** to an operator.

An account-hash-update-transaction contains the old and the updated account hashes, and the associated account-hash-update-proof.

4.1.2 Step 2: money order registration

An operator receives and verifies a batch of account-hash-update-transactions. In other words, the operator verifies the account-hash-update-proofs against the provided pairs of account hashes. Next it

- batches the valid sender transactions,
- executes the batch and computes the new **state-root**,
- and generates a (rollup-)**state-root-update-proof**.

It then creates a blockchain transaction to update the state-root on chain. This transaction contains the batch of executed transactions, the updated state-root-hash and the associated state-root-update-proof. Note that the state-root-update-proof involves **proof recursion** (it is a proof of proofs), see section 4.6.

4.1.3 Step 3: on-chain state-hash update

The operator generated state-root-hash-update-transaction gets picked up by a miner and included on chain. In the process:

- the rollup smart contract verifies the state-root-update-proof found in the transaction,
- upon successful verification, the rollup smart contract updates the on-chain state-root-hash.

4.1.4 Step 4: money-order-inclusion-proof

The sender monitors updates made to the on-chain state-root-hash and awaits the on-chain inclusion of its account-hash-update. Once its transaction has been included it starts the next step in the process.

The sender waits for a random number of rollup state updates. After that, it creates a **money-order-inclusion-proof**. This is a ZKP proving the membership of the previously created money order in the state. The underlying computation is a Merkle-proof. Using a zero-knowledge proof allows the sender to hide the path from the money order to the state-root, in particular we don't want the sender to reveal its $\mathsf{account}_{\mathsf{id}}$.

The sender then sends a **money-order-inclusion-proof** to the receiver by sending a money order receipt message.

4.1.5 Step 5: money order reception

Upon reception of the money order and the associated money-order-inclusion-proof, the receiver checks that:

- the money-order-inclusion-proof is valid,
- the money order is destined to its account,
- check that no one ever communicated money orders with the same hash.

4.1.6 Step 6: money order redemption

To redeem a money order the receiver:

- increases its balance by X tokens,
- updates its randomness,
- inserts the money order in its set of redeemed money orders,
- computes its new account-root-hash.

The receiver then needs to get its account-hash update into the rollup state. To that end, it generates a **money-order-redemption-proof**. This proof attests to the fact that a money order compatible with the proposed account update exists, that an associated money-order-inclusion-proof verifies that money order against a historical state-root-hash, and that the associated account-hash update takes the current account-hash to the updated value.

It then sends the account hash update and the accompanying money-order-redemption-proof to an operator.

4.2 Hiding the link between the sender and the receiver

After the sender's money order has been successfully added to the rollup state, the sender needs to prove to the receiver the existence of a money order. This is done by providing the receiver with a Merkle-proof of inclusion of the money order in the sender's account. Thus,

- the sender queries the operator for a Merkle-path to its account (the root hash is $\mathsf{stateRoot}_0$),
- the sender waits some amount of time, at which point the state hash is $\mathsf{stateRoot}_a$.

This Merkle-proof is then sent over the wire directly from the sender to the receiver. The receiver queries an operator to get a proof of inclusion of the money order in $\mathsf{stateRoot}_a$.

In the process, the receiver gets knowledge of the sender's $\mathsf{account}_{\mathsf{id}}$ but this information is hidden to the operator. The operator only sees $\mathsf{stateRoot}_a$ and cannot learn which money order has been redeemed nor in which block it was created.

4.3 Batching techniques

A user can create up to 2^8 money orders in a single account update; similarly, a user can redeem up to 2^8 money orders in a single account update. These two operations need to be performed separately.

4.4 Historical root hash

The historical root hash stores the list of the 2^{24} last state root hashes. This allows users to access any one of the last 2^{24} deprecated states of the rollup.

4.5 Reset the sent money order Merkle-tree

The sent money order Merkle-tree is reset at every account update. Due to the inclusion of the historical root hash, this data is *not lost*. As a consequence, a relatively small Merkle-tree is enough to allow for many money order issuances.

4.6 Proof recursion

Informally, proof recursion is the act of "proving that another proof has been correctly verified". Several strategies exists to achieve recursion. [3] proposes using pairing-friendly cycles of elliptic curves. [5] relies on Cock-Pinch curves and uses a single-level of recursion. For this work, we use the approach of [5] and use the curves BLS12-377 and SW6 ladder of pairing-friendly elliptic curve.

5 Data structures

Let SMT-x stand for a Sparse Merkle Tree of depth x.

5.1 State

The state is managed by the operators.

Attribute	Type	Description
accounts	SMT-24[AccountHash]	Contains the hash of all accounts.
previousStates	SMT-24[State]	Contains the previous states of the rollup. It is filled sequentially. When the tree reaches capacity, it loops back to the first leaf and overwrites the oldest historical state roots with newer ones. This ensures the previous state roots stay accessible to the rollup for some duration. Any transaction taking longer than this duration can never be completed.

5.2 Account

The account is managed by the users. The operators and the observers do not access this data.

Attribute	Type	Description
accountData	AccountData	See below.
registeredMoneyOrder	SMT-8[MO]	The append-only list of money orders produced by this account in this update. It is reset at every account hash update.
redeemedMoneyOrders	SMT-64[MO]	The insert-only list of redeemed money orders.

5.3 Account data

Account data contains plain text values that should not be disclosed when the sender creates a Merkle-proof of inclusion of a specific money order.

Attribute	Type	Description
balance	int	Balance of the account.
publicKey	int	The public key of the account owner.
randomness	int	Some secret randomness to hide the account data. Updated with every account hash update.

5.4 Money order

A money order is issued by a sender and gives a receiver the permission to credit their account with a specified amount of tokens. Only the participants in the transaction should have access to it. The receiver can only redeem money orders that have been registered.

Attribute	Type	Description
to	int	Receiver of the money order.
amount	int	Amount of token transferred.
id	int	Unique id for each transaction. $\mathsf{id} = \mathsf{hash}(pub_{\mathsf{sender}}, \mathsf{nonce}, \mathsf{salt})$.

5.5 Money order receipt message

Message sent by the sender to the receiver.

Attribute	Type	Description
moneyOrder	MoneyOrder	Money order this receipt is justifying.
moneyOrderInclusionProof	zkp	A ZKP proving the registration of the money order to hide the sender account from the receiver.

5.6 Money order registration message

Message sent by the sender to the operators to register a new batch of money orders.

Attribute	Type	Description
accountId	int	Position of the account to be updated in the Merkle-tree
oldAccountHash	byteArray	Hash of the account before the transition.
newAccountHash	byteArray	Hash of the account after the transition
accountHashUpdateProof	zkp	A ZKP proving the newAccountHash is the result of a money order registration.

5.7 Money order redemption message

Message sent by the receiver to the operators to redeem a batch of money orders.

Attribute	Type	Description
accountId	int	Position of the account to be updated in the Merkle-tree
oldAccountHash	byteArray	Hash of the account before the transition.
newAccountHash	byteArray	Hash of the account after the transition.
accountHashUpdateProof	zkp	A ZKP proving the newAccountHash is valid.
inclusionProof	zkp	The ZKP provided by the sender to prove registration of the money order on chain.
moneyOrderHash	byteArray	The hash of the money order to be redeemed.
stateRootHash	byteArray	State hash given by S as part of the receipt.

6 Operations

6.1 Sender: creation and registration of a money order

The sender inserts a new batch of money order in its $\mathsf{registeredMoneyOrder}$ Merkle-tree $\mathsf{MT}[\mathsf{MO}_0, \mathsf{MO}_1, \dots]$. Each money order consists of data structure $\mathsf{MO}_k = (\mathsf{amount}_k, \mathsf{pubkeyR}_k, \mathsf{id}_k)$ where id_k is defined as $\mathsf{id}_k = \mathsf{hash}(\mathsf{pubkeyS}_k, \rho_k)$ where ρ_k is a unique randomness.

The sender then updates its account in the following way:

- erases the $\mathsf{registeredMoneyOrder}$ Merkle-tree by resetting every leaf to $\mathsf{0x000}\dots$,
- inserts sequentially the money order hashes of the batch, $\mathsf{MO}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{MO})$,
- updates its balance,
- updates its randomness
- computes the new $\mathsf{accountS}_{\mathsf{hash}}$'.

The sender then computes a proof of knowledge for the state update with the following checks:

$$\mathsf{EddsaVerify}(\mathsf{pubkey}, \mathsf{MO}_{\mathsf{hash}}, \mathit{sig})$$
$$\mathit{Range32bits}(\mathsf{balance} - \mathsf{amount})$$
$$\mathsf{MO}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{amount}, \mathsf{pubkey}_R, \mathsf{MO}_{\mathsf{id}})$$
$$\mathsf{MO}_{\mathsf{id}} = \mathsf{hash}(\mathsf{pubkey}_S, \mathsf{nonce}, \mathsf{salt})$$
$$\mathsf{registeredMO}'_{\mathsf{root}} = \mathsf{rootHash}(\mathsf{MO}_0, ...)$$
$$\mathsf{balance}' = \mathsf{balance} - \mathsf{amount}$$
$$\mathsf{accountData}'_{\mathsf{hash}} = \mathsf{hash}(\mathsf{pubkey}_S, \mathsf{balance}'_S, \mathsf{salt}')$$
$$\mathsf{accountData}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{pubkey}_S, \mathsf{balance}_S, \mathsf{salt})$$
$$\mathsf{account}'_{\mathsf{hash}} = \mathsf{hash}(\mathsf{registeredMO}'_{\mathsf{root}}, \mathsf{redeemedMO}_{\mathsf{root}}, \mathsf{accountData}'_{\mathsf{hash}})$$
$$\mathsf{account}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{registeredMO}_{\mathsf{root}}, \mathsf{redeemedMO}_{\mathsf{root}}, \mathsf{accountData}_{\mathsf{hash}})$$

The public inputs of the proof are $\mathsf{account}_{\mathsf{hash}}, \mathsf{account}'_{\mathsf{hash}}$. S proceeds to send the newly constructed **money-order-registration-message**.

6.2 Operator: aggregation of money order registration

When the operator completes a batch of **money-order-registration-messages**, it updates the state in the following way:

- inserts the current rollup state root in the next position in the historical state tree (potentially overwriting the oldest deprecated state contained therein)
- updates the $\mathsf{account}_{\mathsf{hash}}$ for each sender-proof,
- updates the global root hash.

After updating the state, the operator issues two zero-knowledge proofs.

First a proof that the rollup's global root-hash update is consistent with the global state update. The **state-root-update-proof**:

$$\mathsf{rollup}_{\mathsf{root}} = \mathsf{hash}(\mathsf{historicalStatesRoot}, \mathsf{accountsRoot})$$
$$\mathsf{historicalStatesRoot} = \mathsf{MerkleProofCheck}(\mathsf{historicalStatesRoot}[n \,\%\, 2^{24}], \mathsf{path}(\mathsf{currentPositionToOverwrite}), \mathsf{branch})$$
$$\mathsf{historicalStatesRoot}' = \mathsf{MerkleProofCheck}(\mathit{stateRoot}, \mathsf{path}(\mathsf{currentPositionToOverwrite}), \mathsf{branch})$$
$$\mathsf{currentAccounts}_{\mathsf{root}} = \mathsf{accountsRoot}$$

Then, for each account:

$$\mathsf{currentAccounts}_{\mathsf{root}} = \mathsf{MerkleProofCheck}(\mathsf{account}_{\mathsf{hash}}, \mathsf{path}(\mathsf{account}_{\mathsf{id}}), \mathsf{branch})$$
$$\mathsf{currentAccounts}'_{\mathsf{root}} = \mathsf{MerkleProofCheck}(\mathsf{account}'_{\mathsf{hash}}, \mathsf{path}(\mathsf{account}_{\mathsf{id}}), \mathsf{branch})$$

The public parameters are the batch number n, *stateRoot*, *stateRoot*'.

Secondly, a top-level proof which aggregates together all the proof of registration, the receivers proofs and the **state-root-update-proof**. This curve is defined over the second layer of the recursive chain of elliptic curves as described in 4.6.

$$\mathsf{Groth16Verify}((\mathsf{rollup}_{\mathsf{root}}, \mathsf{rollup}'_{\mathsf{root}}, \mathsf{account}^0_{\mathsf{hash}}, \ldots \mathsf{account}^n_{\mathsf{hash}}, \mathsf{account}'\,{}^0_{\mathsf{hash}}, \ldots, \mathsf{account}'\,{}^n_{\mathsf{hash}}), \mathsf{proof}_{\mathsf{operator}}, \mathsf{vk}_{\mathsf{operator,sender}})$$

Then, for each account:

$$\mathsf{Groth16Verify}((\mathsf{account}^i_{\mathsf{hash}}, \mathsf{account}'\,{}^i_{\mathsf{hash}}), \mathsf{proof}_i, \mathsf{vk}_{\mathsf{sender}})$$

And we assign $\mathsf{currentAccounts}_{\mathsf{root}} = \mathsf{currentAccounts}'_{\mathsf{root}}$ at each iteration.

Then again, only once after each $\mathsf{currentAccounts}_{\mathsf{root}}$ update $\mathsf{rollup}'_{root} = \mathsf{hash}(\mathsf{accountsRoot}', \mathsf{historicalStatesRoot}')$.

6.3 Sender: communication of the money order receipt to the receiver

Let $\mathsf{stateRoot}_0$ denote the rollup root hash at the moment the money order is registered. The sender waits for a duration of its choice until the state root of the rollup is $\mathsf{stateRoot}_a$. The sender then creates a **money-order-inclusion-proof**:

$$
\begin{aligned}
&\mathsf{stateRoot}_0 = \mathsf{hash}(\mathsf{historicalStates}_0, \mathsf{accountsRoot}_0)\\
&\mathsf{stateRoot}_a = \mathsf{hash}(\mathsf{historicalStates}_a, \mathsf{accountsRoot}_a)\\
&\mathsf{historicalStates}_a = \mathsf{MerkleProofCheck}(\mathsf{stateRoot}_0, a_{\mathsf{branch}}, a_{\mathsf{path}})\\
&\mathsf{accountsRoot}_0 = \mathsf{MerkleProofCheck}(\mathsf{accountS}_{\mathsf{hash},0}, \mathsf{branch}_{\mathsf{stateRoot}_0}, \mathsf{accountS}_{id}])\\
&\mathsf{accountS}_{\mathsf{hash},0} = \mathsf{hash}(\mathsf{registeredMO}_{\mathsf{root},0}, \mathsf{redeemedMO}_{\mathsf{root},0}, \mathsf{accountDataS}_0)\\
&\mathsf{redeemedMO}_{\mathsf{root},0} = \mathsf{MerkleProofCheck}(\mathsf{MO}_{\mathsf{hash}}, \mathsf{MO}_{\mathsf{path}}, \mathsf{MO}_{\mathsf{branch}})\\
&\mathsf{MO}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{amount}, tmp_{\mathsf{R}}, \mathsf{id})
\end{aligned}
$$

Where the public inputs are $\mathsf{MO}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{amount}, \mathsf{pubkey}_R, \mathsf{id})$ and $\mathsf{stateRoot}_a$. This is a big Merkle-proof of inclusion wrapped in a SNARK.

The message sent to the receiver is $(\mathsf{amount}, \mathsf{pubkey}_R, \mathsf{id}, \mathsf{stateRoot}_a, \mathit{inclusionProof})$. The receiver uses these values to recompute $\mathsf{MO}_{\mathsf{hash}}$ by itself. The receiver must check the validity of the messages by verifying the proof.

An important feature of this zero-knowledge proof is that it does not leak information about $\mathsf{stateRoot}_0$, the state of the rollup when the MO was created. This in turn would leak information about which batch of transactions contained the MO creation.

6.4 Receiver: redemption of money orders

On reception of one or multiple valid money order receipts from one or multiple senders, the receiver can update its account state and generate the associated money-order-redemption-proof. The receiver:

- checks the money-order-inclusion-proof is valid,
- checks the money order is destined to its account,
- checks that no one ever communicated money orders with the same hash,
- increases its account balance by the total amount specified by the money orders,
- inserts the money orders ids in the sparse Merkle-tree redeemedMO at the position defined by the 64 firsts bits of the $\mathsf{MO}_{\mathsf{id}}$.
- updates its account randomness.

The **money-order-redemption-proof** contains the following checks for each money-order-receipt:

$$
\begin{aligned}
&\mathsf{MO}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{amount}, \mathsf{pubkey}_{\mathsf{R}}, \mathsf{MO}_{\mathsf{id}})\\
&\mathsf{accountR}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{accountDataR}_{\mathsf{hash}}, \mathsf{registeredMO}_{\mathsf{root}}, \mathsf{redeemedMO}_{\mathsf{root}})\\
&\mathsf{accountR}'_{\mathsf{hash}} = \mathsf{hash}(\mathsf{accountDataR}'_{\mathsf{hash}}, \mathsf{registeredMO}_{\mathsf{root}}, \mathsf{redeemedMO}'_{\mathsf{root}})\\
&\mathsf{balanceR}' = \mathsf{balanceR} + \mathsf{amount}\\
&\mathsf{accountDataR}_{\mathsf{hash}} = \mathsf{hash}(\mathsf{pubkey}_{\mathsf{R}}, \mathsf{balanceR}, \mathsf{salt})\\
&\mathsf{accountDataR}'_{\mathsf{hash}} = \mathsf{hash}(\mathsf{pubkey}_{\mathsf{R}}, \mathsf{balanceR}', \mathsf{salt}')\\
&\mathsf{MO}_{\mathsf{path}} = \mathsf{unpack}(\mathsf{MO}_{\mathsf{id}})[..64]\\
&\mathsf{redeemedMO}_{\mathsf{root}} = \mathsf{MerkleProofCheck}(0x0000\ldots, \mathsf{MO}_{\mathsf{path}}, \mathsf{MO}_{\mathsf{branch}})\\
&\mathsf{redeemedMO}'_{\mathsf{root}} = \mathsf{MerkleProofCheck}(\mathsf{MO}_{\mathsf{id}}, \mathsf{MO}_{\mathsf{path}}, \mathsf{MO}_{\mathsf{branch}})
\end{aligned}
$$

The public inputs of the money-order-redemption-proof are the $\mathsf{MO}_{\mathsf{hash}}$, $\mathsf{stateRoot}_a$, $\mathsf{accountR}_{\mathsf{hash}}$ and $\mathsf{accountR}'_{\mathsf{hash}}$

6.5 Operator: aggregation of money order redemptions

When an operator completes a batch of redemption proofs, it:

- inserts the current rollup Root at its corresponding position by overwriting the oldest *stateRoot* in the historicalStatesMT,
- updates the $\mathsf{account}_{\mathsf{hash}}$ for each receiver,
- updates the global root hash.

The operator receives multiple tuples $(\mathsf{proof}, \mathsf{account}_{\mathsf{id}}, \mathsf{account}'_{\mathsf{hash}}, \mathsf{stateRoot}_a)$ and issues a **state-root-update-proof**.

$$\begin{aligned}
&\mathsf{rollup}_{\mathsf{root}} = \mathsf{hash}(\mathsf{accountsRoot}, \mathsf{historicalStatesRoot})\\
&\mathsf{historicalStatesRoot} = \mathsf{MerkleProofCheck}(\mathit{oldRoots}[n \,\%\, 2^{24}], \mathsf{path}(\mathit{currentPositionToOverwrite}), \mathsf{branch})\\
&\mathsf{historicalStatesRoot}' = \mathsf{MerkleProofCheck}(\mathsf{rollup}_{\mathsf{root}}, \mathit{path}(\mathit{currentPositionToOverwrite}), \mathsf{branch})\\
&\mathsf{currentAccounts}_{\mathsf{root}} = \mathsf{historicalStatesRoot}
\end{aligned}$$

Then, for each account:

$$\begin{aligned}
&\mathsf{MerkleProofCheck}(\mathsf{account}_{\mathsf{hash}}, \mathsf{path}(\mathsf{account}_{\mathsf{id}}), \mathsf{branch}) = \mathsf{currentAccounts}_{\mathsf{root}}\\
&\mathsf{MerkleProofCheck}(\mathsf{account}'_{\mathsf{hash}}, \mathsf{path}(\mathsf{account}_{\mathsf{id}}), \mathsf{branch}) = \mathsf{currentAccounts}'_{\mathsf{root}}\\
&\mathsf{MerkleProofCheck}(\mathsf{stateRoot}_a, a_{\mathsf{path}}, a_{\mathsf{branch}}) = \mathsf{historicalStatesRoot}'
\end{aligned}$$

At each iteration assign $\mathsf{currentAccounts}_{\mathsf{root}} = \mathsf{currentAccounts}'_{\mathsf{root}}$
Then again, only once after each $\mathsf{currentAccounts}_{\mathsf{root}}$ update

$$\mathsf{rollup}'_{\mathsf{root}} = \mathsf{hash}(\mathsf{historicalStatesRoot}', \mathsf{accountsRoot}') \tag{1}$$

Then again, the operator generates an aggregation proof of all the inclusion proofs, redemption proofs and the state-update proofs:

$$\begin{aligned}
&\mathsf{Groth16Verify}((\mathsf{rollup}_{\mathsf{root}}, \mathsf{rollup}'_{\mathsf{root}}, \mathsf{account}^0_{\mathsf{hash}}, \ldots \mathsf{account}^n_{\mathsf{hash}};\\
&\mathsf{account}'_{\mathsf{hash}}{}^0, \ldots, \mathsf{account}^{n'}_{\mathsf{hash}}), \mathsf{proof}_{\mathsf{state\text{-}update}}, \mathsf{vk}_{\mathsf{operator,receiver}});\\
&\mathsf{Groth16Verify}((\mathsf{MO}_{\mathsf{hash}}, \mathsf{stateRoot}_a), \mathsf{proof}_{\mathsf{inclusion}}, \mathsf{vk}_{\mathsf{inclusion}});\\
&\mathsf{Groth16Verify}((\mathsf{MO}_{\mathsf{hash}}, \mathsf{stateRoot}_a, \mathsf{account}_{\mathsf{hash}}, \mathsf{account}'_{\mathsf{hash}}), \mathsf{proof}_{\mathsf{redemption}}, \mathsf{vk}_{\mathsf{redemption}});
\end{aligned}$$

References

[1] HarryR Yondon Fu Philippe Castonguay BarryWhitehat, Alex Gluchowski. Roll-up and roll-back side-chain with around 17000 transactions per seconds. https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675, 2019.

[2] Vitalik Buterin. On-chain scaling at potentially 500 transactions per seconds. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477, 2019.

[3] Eran Tromer Eli Ben-Sasson, Alessandro Chiesa and Madars Virza. Scalable zero knowledge via cycles of elliptic curves, 2014.

[4] Alexandre Belling Olivier Bégassat and Nicolas Liochon. Account-based anonymous rollup. https://ethresear.ch/t/account-based-anonymous-rollup/6657, 2019.

[5] Matthew Green Ian Miers Pratyush Mishra Howard Wu Sean Bowe, Alessandro Chiesa. Zexe: Enabling decentralized private computation.