Partially anonymous rollups

Olivier Bégassat, Alexandre Belling, Nicolas Liochon

v1.6, March 2022

Abstract

This note contains the specification of **partially anonymous rollups** — a rollup design with anonymity and scalability properties halfway between those of a zk-rollup [1, 2] and those of a fully anonymous zk-rollup as specified in [3]. With partially anonymous rollups, the operator creating a batch has access to the transaction details before executing it. The protocol we propose here is account based and allows for a very high transaction throughput. Performances do not degrade with the number of transactions previously executed. However, account activity leaks globally in the form of account hash updates.

Contents

1 Introduction

Partially anonymous rollups are a rollup[1] design that sits halfway between standard (fully transparent) rollups and fully anonymous rollups. In a rollup, the rollup state is managed off chain, the rollup smart contract only knows the rollup's state root hash (SRH). State updates involve operators updating said SRH. How this is done in practice differs from one rollup design to another. Among the variables that differentiate these rollup designs one finds:

User/Operator communication: how users request operators perform transactions on their behalf, in particular what transaction details need to be revealed to operators and/or the wider network.

Operator/Blockchain communication: how operators justify updates to the rollup SRH, and how much account activity is leaked in the process.

In a *standard (transparent) rollup*, users send their desired transactions *transparently* to an operator, and the operator justifies SRH transitions associated to a batch of transaction with a proof of computational integrity. In particular, *full transaction details* are published as part of the transaction data of the blockchain transaction realizing the SRH update. On chain transparency is necessary for other operators to update their view of the rollup state. In a *fully anonymous rollup*, transaction secrecy and anonymity is achieved through the means of user generated zero knowledge proofs. In other words, users do not provide operators with transaction details (such as issuing account, recipient account and transfer amount), rather they provide operators with a proof of a valid account transition justified against (an account root hash and ultimately) the rollup SRH. Operators accordingly construct proofs of proofs, batching together proof verifications for account (hash) updates and never achieve any insight into the actual state of the rollup. Thus, anonymous rollups achieve full anonymity in the sense that state updates of the rollup leak no information (neither to operators nor on chain) about the identities of the participants of a transaction nor about the funds being transfered. Even users *redeeming* a money order[2] will not know which account issued the money order they are redeeming.

The approach for partially anonymous rollups we present here relaxes the strong anonymity requirements that are at the core of our fully anonymous rollup proposal [3] while preserving some anonymity. In essence, users communicate transparently with operators, operators communicate SRH updates on chain (and thus to other operators) only through *hashes* and *encrypted data*: updated account state *hashes*, money order *hashes*, updated state root *hashes* and *encrypted data* accompanying these transactions.

	User/Operator communication	Operator/Blockchain communication
Standard rollup	transparent	transparent
Anonymous rollup	**obfuscated**	**obfuscated**
Partially anonymous rollup	transparent	**obfuscated**

Both our fully anonymous and partially anonymous rollups use the **money order** paradigm: transactions from one party to another are done in two steps: **money order creation** (i.e. burning funds from the issuer's rollup account and insertion of a transaction digest, the money order hash, into a dedicated sparse Merkle tree (SMT)) and **money order redemption** (i.e. the opening of a committed money order hash and claiming of the funds locked therein).

[1]Throughout this document, rollup = zk-rollup, i.e. a rollup where state transitions are justified with a zero knowledge proof. The only kinds of rollup we consider are standard (zk) rollups, partially anonymous rollups and anonymous rollups.

[2]We explain money orders below.

2 State

2.1 The state — operator view

The partially anonymous rollup state, as witnessed by an operator, is comprised of the following parts:

1. the **account tree**,
2. the **money order tree**.

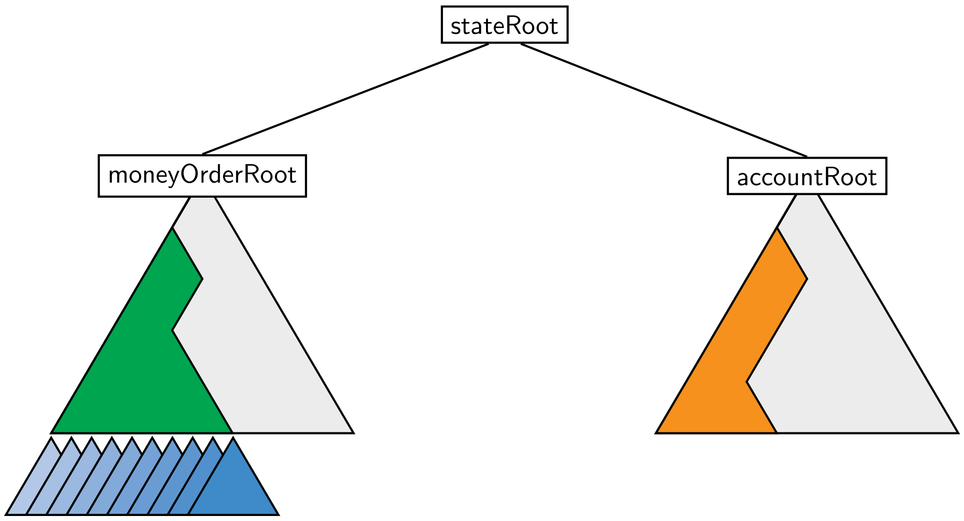

Figure 1: The money order SMT and account SMT. The leaves of the money order SMT are the root hashes of money order batches (represented by blue triangles). Money order batches contain money orders. Money orders are created either by money order creation (i.e. intra rollup transactions) or inbound transfers (i.e. outside funds entering the rollup).

Both the account tree and the money order tree are 2-ary SMTs of depth 32 in append mode whose leaves contain field elements (32 byte integers). Both trees are stored in memory. The leaves of the account tree contain **account hashes**, i.e. the hashes of an auxiliary data structure called an **account** which we explain below. Operators know the contents of the accounts they themselves manage - these accounts are also kept in memory - of the others they only know their hash. The leaves of the money order tree contain **money order batch money order hashes**, which we also explain below. Both of these SMTs are updated continuously as updates roll in. This involves inserting new leaves in either tree or updating existing leaves in the account SMT; in both cases it further involves updating all relevant Merkle paths and in particular updating the root hashes of both SMTs.

Operators may store (on disk) certain batches of transaction hashes. To be precise, the encrypted data accompanying batches of transactions allows operators (and relevant users) to recognize those transaction hashes that involve some of their users. Depending on whether a batch contains a transaction destined to one of their users or not they may choose to remember said batch of transaction hashes (or even just the associated Merkle paths) as the associated Merkle paths are necessary for later redemption.

All this data can be reconstructed from on transaction data (transaction hashes, account hashes and encrypted data). Users can also (if need be) reconstruct the state of their account using transaction data.

2.2 The state — on-chain view

The smart contract of a partially anonymous rollup contains a single root hash, the “state money order hash” (abbreviated to SRH) representing the totality of the rollup’s state. This root hash is in fact the hash

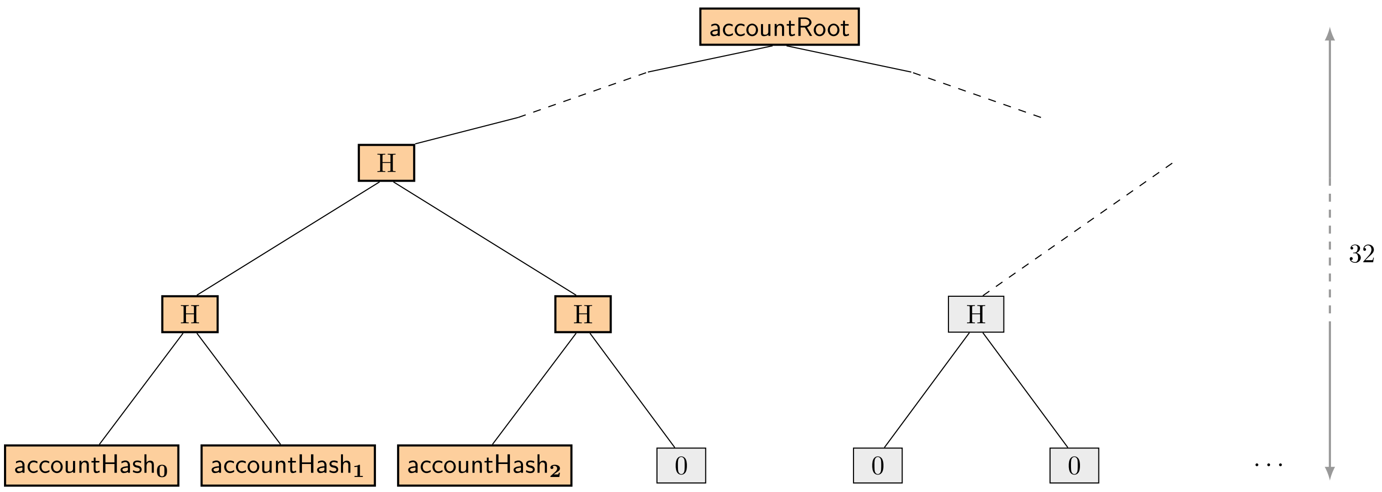

Figure 2: Every nonempty leaf of the account SMT contains the account hash of the account with ID equal to the position of the leaf in the tree. In the above example there are 3 existing accounts.

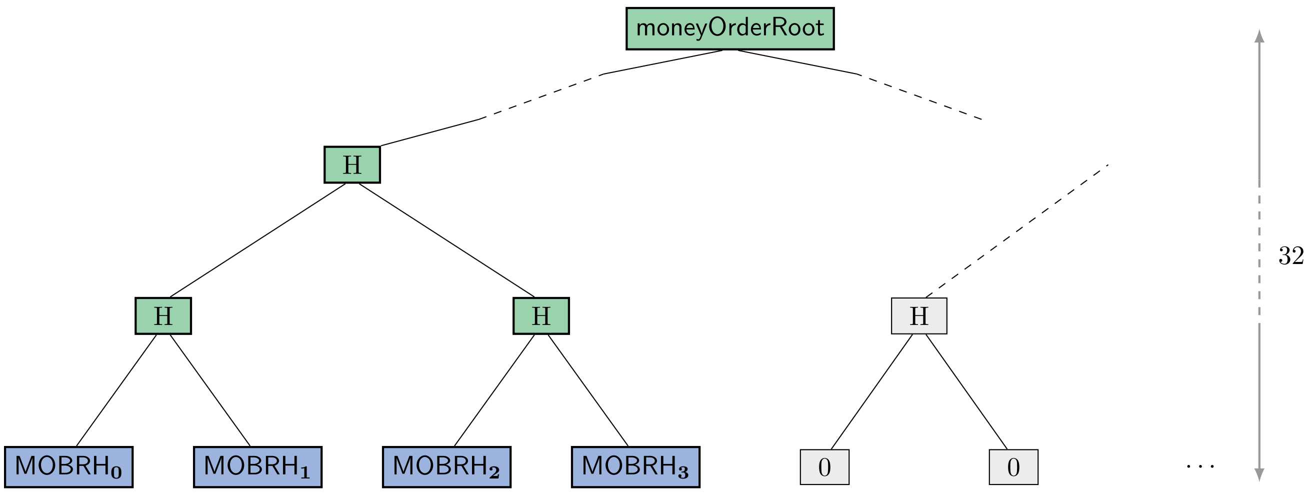

Figure 3: Every nonempty leaf of the money order SMT contains a money order batch money order hash (abbreviated MOBRH, details explained below). In the diagram above the state contains 4 money order batches.

of the account SMT root hash and the money order SMT root hash:

$$\textsc{srh} = \mathsf{Hash}\left[\text{money order } \textsc{smt} \text{ money order hash} \,\middle\|\, \text{account } \textsc{smt} \text{ money order hash}\right]$$

Transaction batches sent to the partially anonymous rollup smart contract update the SRH by implicitely updating the account SMT or the money order SMT (or both).

3 Accounts, account hashes and the account SMT

3.1 Accounts

Within a partially anonyous rollup, an **account** is fully described by the following data:

accountId	4 byte integer	*public*
publicKey	elliptic curve point	*public*
jointEncryptionKey	2 elliptic curve points	*public*
nonce	4 byte integer	*private*
redemptionIndex	6 byte integer	*private*
blindingFactor	1 field element	*private*
balanceRoot	1 field element	*private*
balances	slice of 8 byte integers	*private*

(The user of course knows the secret key associated with the account's public key.)

accountId: the position occupied by the account in the account SMT. The accountId isn't part of the account *per se* (we don't include it when hashing the account for instance). It is *public*.

publicKey: used to verify signatures justifying the authorship of any requested transaction. It is *public*.

jointEncryptionKey: used for data availability: all transaction details necessary for (1) users to recontruct the state of their account and be able to recognize incoming transactions targeted at them and (2) for operators to be able to reconstruct the state of their managed accounts as well as be aware of incoming transactions directed at their users — all of it is encrypted using the user's jointEncryptionKey (and the recipient's jointEncryptionKey if it applies) and posted on chain. It is *public*.

nonce: integer initialized at 0 at account creation and augmented by one with every successful account operation. It can be recomputed from *public* data.

redemptionIndex: integer initialized at 0 and set to the last moneyOrderTotalIndex after every successful money order redemption; the moneyOrderTotalIndex represents the id of a money order in money order tree augmented by the money order batches (i.e. viewed as a tree of depth $32 + 16 = 48 = 6 \times 8$. It is *private*.

blindingFactor: random field element set at account creation and used to obfuscate the contents of the account. It is *private*.

balances: slice of 2^{16} 64 bit integers representing the balances of the tokens held by the account (each token is associated with a unique ID in the range $[\![0, 2^{16} [\![$.) They are *private*.

balanceRoot: the root hash of the Merkle tree of depth 16 build from the balances slice. It is *private*.

(Note that the balanceRoot can be derived from the balances slice; we view it as a field of the account for sheer convenience.)

At account creation the nonce, redemptionIndex and balances are set to 0. The nonce plays the usual role of ordering transactions and ensuring they get executed only once, and is incremented with every operation triggered by the account. We explain the redemptionIndex in subsection 7.4. Its purpose is to prevent double redeems of money orders; it further allows our scheme not to include nullifiers for money orders. The publicKey and blindingFactor is decided by the account holder (either an Ethereum address requesting a rollup account creation on chain or someone requesting an account creation directly to an operator). The **accountId** is set incrementally at account creation. It is the position within of the account hash in the account SMT. The jointEncryptionKey is defined by the operator at account creation. It is constructed from the operator's public key and the account's public key.

3.2 Account hashes

Given an account, its accountHash is the hash of the fields of the account in a specified order:

$$\mathsf{accountHash} = \mathsf{Hash}\big(\big[\mathsf{jointEncryptionKey} \parallel \mathsf{publicKey} \parallel \mathsf{accountStateHash}\big]\big)$$

where the accountStateHash itself is a hash:

$$\mathsf{accountStateHash} = \mathsf{Hash}\big(\big[\mathsf{blindingFactor} \parallel \mathsf{redemptionIndex} \parallel \mathsf{nonce} \parallel \mathsf{balanceRoot}\big]\big)$$

Recall that the balanceRoot is itself the root hash of the Merkle tree whose leaves are the account's balances slice.

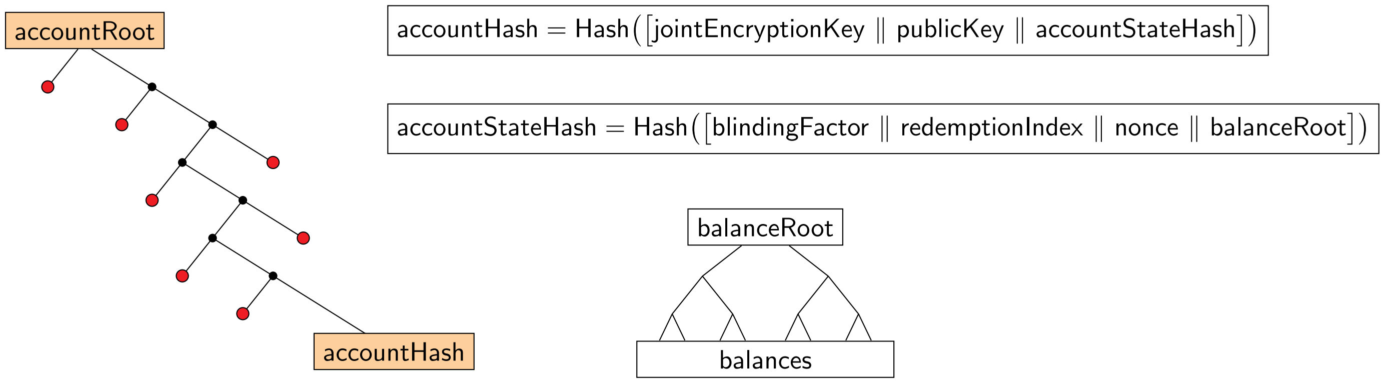
accountRoot
accountHash = Hash([jointEncryptionKey ∥ publicKey ∥ accountStateHash])
accountStateHash = Hash([blindingFactor ∥ redemptionIndex ∥ nonce ∥ balanceRoot])
balanceRoot
accountHash
balances

Figure 4: Representation of the internals of an account and the nested structure of the accountHash along with the Merkle Path linking the accountHash to the account SMT root hash.

3.3 The account tree

The account SMT is a depth 32 arity-2 SMT whose leaves are indexed by accountId (ranging from 0 to $2^{32} - 1$). Each leaf contains the accountHash of the account with that accountId. Leaves with an accountId greater than the current greatest allocated accountId contain 0. Along with the money order SMT, the account SMT is one of two SMTs operators keep in memory at all times and continuously update as state updates roll in. Updating the account SMT means:

1. inserting new leaves as they are created through account creations,
2. updating the account hashes of existing accounts as the accounts are updated either through money order creation, money order redemption, inbound transfers and outbound transfers.

Every modification of the leaves requires updating the remainder of the SMT (i.e. nodes at higher levesl), in particular the accountRoot. The account SMT starts out empty (i.e. all its leaves contain 0). Each (batch) account creation inserts new accounts linearly (i.e. in increasing order of accountIds starting from the first available accountId).

The account SMT can hold up to $2^{32} \simeq 4.3 \cdot 10^9$ accounts. When the account SMT is full there is no way to add further accounts. If more accounts are needed one can use a deeper account SMT from the very start. Doubling the number of available accounts adds a single hash to every Merkle proof verifying an account hash against the account root. It does not double the memory requirements on the operator side, by virtue of the SMT-based design: the memory is used only when the accounts are actually created.

3.4 Account funding

There is only *one* way to fund an account: through money order redemptions. There are, however, *two* ways to create money orders for an account to redeem: (1) external funds enter the rollup as money orders created through an **inbound transfer** and (2) funds are moved within the rollup through **money order creation**. Funds can only be redeemed by an existing rollup account. In any case creating and funding an account is a two step process: account creation followed by money order redemption.

3.5 Account management

Any rollup operations gets funneled through an operator who is thereby made aware of some (or all) of the fields of the account. Operators (as well as anybody monitoring the rollup smart contract) are aware of every account's publicKey and jointEncryptionKey, as these are made public at **account creation**. One may also compute an account's nonce by tallying its historical updates. Operators are furthermore aware of all the private fields of the accounts they manage. When accounts are updated the transaction contains the updated accountStateHash of every updated account, i.e. the hash of the private portion of the account. From the updated accountStateHash and the account's **publicKey** and **jointEncryptionKey** all operators can derive the updated accountHash, which is then inserted into the account SMT. However only the user's operator is privy to the private fields of the account.

The inclusion of encrypted data (i.e. cipher texts) into the transactions allows users and their operator) to fully reconstruct the state of the user's account. In particular one can recover the blindingFactor (the encrypted version of it is posted at account creation), redemptionNonce, balances and from there the associated balanceRoot.

Account owners may find it convenient to transact through the same operator for an extended period of time. Indeed, operators may provide the service of storing their customer's account (private) data. If such is the case, the account holder need only hold onto their rollup account's private key. While the blinding factor field inside the account can, in theory, be changed with every transaction triggered by the account (e.g. money crder creation/redemption), changing it only when switching operators is sufficient.

4 Money orders and the money order SMT

4.1 Plain money order

The data structure that describes a transaction is called a **plain money order**. A plain money order consists of the transaction details of a transfer of funds between two rollup accounts (or the injection of funds into the rollup through an inbound transfer) along with some random data for obfuscation (a transaction specific **blinding factor**). As such, a plain money order consists of

fromId	4 byte integer	sender account ID
toId	4 byte integer	receiver account ID
tokenId	2 byte integer	token type identifier
amount	8 byte integer	token amount of transaction
txBlindingFactor	field element	blinding factor for obfuscation

The fields fromId and toId are account IDs. The tokenId is for multi-token partially anonymous rollups. Any one plain money order concerns *a single token id*. Thus transfering t different token types from one rollup account requires creating t distinct plain money orders. A plain money orders also contains a txBlindingFactor, a field of random data used for obfuscation (which bears no relation to the user account's blindingFactor). In case of an inbound transfer the fromId and txBlindingFactor are given default values (the 4 byte integer 0 and the 32 byte integer 0 respectively).

4.2 Money order hashes

The fields of a plain money order are sufficient to compute its hash. We define the hash of a plain money order as

$$\mathsf{moneyOrderHash} = \mathsf{H}\big(\big[\mathsf{txBlindingFactor} \| \mathsf{bufferedFields}\big]\big)$$

where $\mathsf{bufferedFields}$ is (4+4+2+8) byte integer

$$\mathsf{bufferedFields} = \mathsf{fromId} \cdot 2^{32+16+64} + \mathsf{toId} \cdot 2^{16+64} + \mathsf{tokenId} \cdot 2^{64} + \mathsf{amount}$$

understood as a field element.

4.3 Money order batches and their root hashes

Money orders created by money order creation are reported on chain through their hash (i.e. as money order hashes)[3]. In the case of a money order created by inbound transfer (i.e. funds entering the rollup) all the fields are public (and given default values in the case of fromId and $\mathsf{blindingFactor}$).

In either case, operators group money orders into **money order batches** containing up to 2^{16} money orders. The associated money order hashes are sequentially inserted into the leaves of a SMT of depth 16 (padded with 0's if the batch isn't full). The root of this money order hash tree is dubbed the **money order batch root hash**.

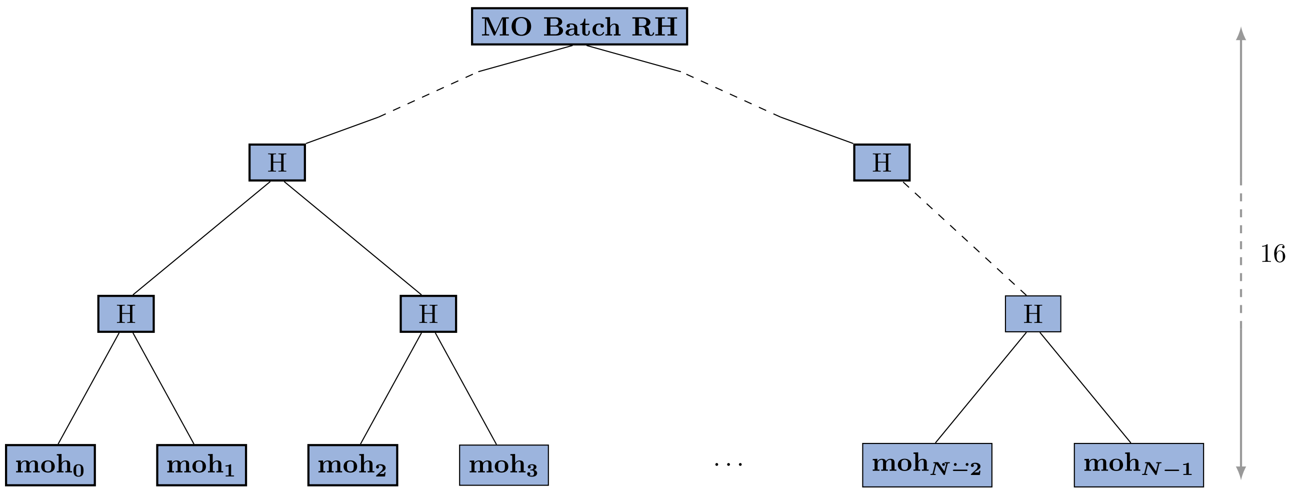

Figure 5: The leaves of a money order batch are up to $N = 2^{16}$ individual money order hashes. The money order batch root hash is what actually gets inserted into the money order SMT.

Money orders batches get included into the money order SMT by inserting the money order batch root hash in the first available slot of the money order tree. The individual money order hashes are included in the transaction data of a batch money order creation transaction (along with the money order batch root hash).

[3]In case of a money order creation the (circuit verified) money order hash is accompanied by

1. the sender id fromId and updated $\mathsf{accountStateHash}$ (both circuit verified)
2. the sender generated encrypted data $(S, B_{\mathsf{snd}}, B_{\mathsf{snd\text{-}op}}, , B_{\mathsf{rec}}, , B_{\mathsf{rec\text{-}op}}, \delta_{\mathsf{mo}}, \delta_{\mathsf{bf}})$ where $(S, B_{\mathsf{snd}}, B_{\mathsf{snd\text{-}op}})$ and $(S, B_{\mathsf{rec}}, B_{\mathsf{rec\text{-}op}})$ (all circuit certified)
3. the encrypted plain money order $\delta_{\mathsf{mo}}, \delta_{\mathsf{bf}}$ (both circuit certified).

Encryption will be explain later.

4.4 The money order SMT

Along with the account SMT, the money order SMT is an arity-2 depth 32 sparse Merkle tree in append mode. Its leaves contain the money order batch root hashes that have been added to the rollup. Operators keep this tree in memory in full as they need to be able to quickly access the Merkle paths linking money order batch root hashes to the money order SMT root hash. Every money order batch Creation transaction and every inbound transfer[4] batch transaction adds a single leaf to the money order SMT. Updating the money order SMT then requires the operators to insert the associated money order batch root hash into the first available slot of the money order SMT and updating its Merkle path, i.e. the Merkle path linking the newly filled slot to the money order SMT root hash.

The money order SMT can hold up to 2^{32} batches of money orders for a total of $2^{32+16} \simeq 2.81 \cdot 10^{14}$ individual money orders (recall that money order batches have a maximum capacity of 2^{16} money orders). Batches don't have to be full, though, so in practice a money order SMT will hold fewer money orders than its maximal capacity. As with the account SMT, when all leaves in the money order SMT are filled up there is no way to add further money order batches to the tree. At a rate of one money order batch per second, it takes roughly 136 years to fill up the tree. One may also choose to make the money order SMT deeper. Every extra layer in the money order SMT adds a single hash to every money order batch insertion or money order batch read. It does not double immediately the amount of memory needed to store the SMT: the memory is needed once the batch has been created.

5 Encryption and data availability

5.1 Purpose of encryption

The purpose of encryption is to convey information to select parties in the rollup (including oneself and one's own operator) to ensure data availability and to inform select other parties of transaction details. Certain operations such as money order redemptions are only of concern to the user and their operator; the details of these operations should be recoverable by either party at any point in time. E.g. for a user to reconstruct their account or for the operator to reconstruct their view of the state. Other operations such as money order creations are of interest to both the issuer, the recipient and their respective operators. The relevant details of these operations should similarly be recoverable by all concerned parties at any point in time. In the case of money order creations there is a further incentive which is that operators (resp. users) ought to be able to identify within money order creation batches posted by other operators those money order creations that concern one of their users (resp. themselves). There are thus three purposes of encryption:

1. encryption for one's own data availability (as well as data availability for one's operator);
2. encryption for another user's data availability (as well as their operator);
3. encryption to communicate to another user (and their operator) the plain money order that was created for them (which they may redeem at a later date).

5.2 Joint encryption key

Every user-operator pair has a public **Joint encryption key** consisting of two elliptic curve points $(M_{\mathrm{u}}, M_{\mathrm{op}})$ on an elliptic curve E with a specified basepoint g. The first one, M_{u}, is user specific and user defined. It has an associated secret key sk_{u} (with $M_{\mathrm{u}} = \mathsf{sk}_{\mathrm{u}} \cdot g$) known only to the user. The second one, M_{op} is the operator's and should be reused for every one of the operator's users. It has an associated secret key $\mathsf{sk}_{\mathrm{op}}$ (with $M_{\mathrm{op}} = \mathsf{sk}_{\mathrm{op}} \cdot g$) known only to the operator.

The joint encryption key can be used by any *user* or *operator* to perform **Elgamal encryption**. Recall that, given an elliptic curve E with chosen basepoint g, the Elgamal encrytpion of a point $A \in E$ using a

[4]See section 6.

public key $\mathsf{pk} = \mathsf{sk} \cdot g$ is the pair (S, B) of elliptic curve points where S and B are points on E computed from pk, g, the point A and a secret scalar s like so:

$$S = s \cdot g \text{ and } B = A + s \cdot \mathsf{pk}.$$

Decryption recovers A by computing $T = \mathsf{sk} \cdot S$ $(= s \cdot \mathsf{pk})$ and deducing A as follows: $A = B - T$.

One should note that *users* will have to do encryption (e.g. when redeeming a money order, the user produces an ecryption according to the first scheme, when creating a money order the issuer produces an encryption according to the second scheme).

The point A (which concerned parties can recover from its respective Elgamal encryption) may later serve to derive an encryption/decryption key κ for a different encryption scheme. In our implementation κ is a scalar obtained by hashing (the coordinates of) A; it is used as the encryption key in the MiMC cipher to encrypt certain fields of a transaction.

5.3 Encryption and decryption for a joint encryption key

Consider a joint encryption key $(M_{\text{u}}, M_{\text{op}})$ as above. Encryption of an elliptic curve point A using a joint encryption key $\mathsf{jek} = (M_{\text{op}}, M_{\text{u}})$ is simply Elgamal encryption of A w.r.t. both public keys (and using the same secret s for both encyrptions): $\mathsf{Enc}_{\mathsf{jek}}(A) = (S, B_{\text{u}}, B_{\text{op}})$ where $S = s \cdot g$, $B_{\text{op}} = A + s \cdot M_{\text{op}}$ and $B_{\text{u}} = A + s \cdot M_{\text{u}}$. Given such an encryption (S, B_1, B_2) both the user and the operator can recover the point A by performing Elgamal decryption on the pairs (S, B_1) and (S, B_2) respectively.

Encryption using a single joint encryption key serves mainly the first purpose: one's own data availability.

5.4 Encryption and decryption for a pair of joint encryption keys

For money order creations the issuer of the transaction encrypts a point A for both their own joint encryption key and that of the receiver. Thus, if we denote by $(M_{\text{snd}}, M_{\text{snd-op}})$ and $(M_{\text{rec}}, M_{\text{rec-op}})$ the joint encryption keys of the sender and the receiver repectively, the user must compute

$$\big(S,\ B_{\mathsf{snd}},\ B_{\mathsf{snd\text{-}op}},\ B_{\mathsf{rec}},\ B_{\mathsf{rec\text{-}op}}\big)$$

where $(S, B_{\mathsf{snd}}, B_{\mathsf{snd\text{-}op}})$ and $(S, B_{\mathsf{rec}}, B_{\mathsf{rec\text{-}op}})$ are the encryptions of a given point A for the respective joint encryption keys using a common secret scalar s.

Encryption a pair of joint encryption key serves all three purposes highlighted above; it is used for money order creations only.

5.5 Decryption of a point with a joint encryption key

There are two scenarios where a user or an operator may try decryption (i.e. Elgamal decryption using the secret key associated to one's half of the joint encryption key):

1. when scanning new money order creation batches to identify money orders destined to oneself (as a user or as an operator acting on behalf of ones users),
2. when scanning historical transaction batches to reconstruct ones state (either one's account as a user or one's view of the state as an operator.)

Recall that operators reuse their half of their joint encryption key with all their users. Thus one attempt at decryption is enough per transaction.

5.6 On verifying encryption

As will become apparent when we describe the circuits for different transaction types *none of the encryptions are checked in circuit.* Encrypted data is always included in the proving circuits as a cipher text and the only contact it makes with the proving circuit is in a signature-verification-subcircuit as part of a (supposedly signed) message. The protocol thus allows users to post incorrect encrypted data (for instance not use the recipient joint encryption key at all, or encrypt data that misrepresents the contents of a transaction). This allows users who don't want to have their data on chain to avoid doing so.

Operators can verify the integrity of the encrypted data and may choose to accept or reject transactions that are incorrectly encrypted. In other words, a user and its operator can together make an account non recoverable from on chain data. Operators, however, cannot lie on behalf of their users by changing the encrypted data: the encrypted data is signed by the user and the signature verified in the circuit.

This design choice is justified by the observation that lying on encrypted data has limited negative effects. Indeed, it has no effect on the transaction taking place. What it can do is:

1. make one's own data unrecoverable beyond a certain point in time to oneself, one's operator or both,
2. in the case of a money order creation: hide the transaction from its recipient, their operator or both.

In both cases, the user that issued incorrectly encrypted data (which they sign) is the one who pays the price (in the first case they either lock themselves or their operator out of data availability, in the second case their funds are spent, but cannot be recovered by the recipient unless they are directly notified by the issuer of the money order). A user may encrypt false data correctly so that yet another user may decrypt incorrect data. However, that user will recognize the deception after recomputing the money order hash from the decrypted data, comparing it to the one posted on chain in the batch - there will be a mismatch.

6 Operations

We discuss the operations supported in a partially anonymous rollup.

Account creation. Creation of a new rollup account, i.e. the insertion a single new account hash into the account SMT at an empty slot. The newly created account starts out with all its balances, nonce and redemption index set to zero. It contains a *user chosen* public key, joint encryption key and blinding factor. The account ID is decided by the operator and lies in interval of length nCreates (the number of account creations in the batch) starting with the first available ID in the account SMT.

Money order creation. Insertion of a money order hash (actually, of the root hash of a money order batch containing it) into the money order SMT. This updates the issuing account SMT accordingly by changing a balance and adding 1 to the nonce.

Money order redemption. Transaction crediting an account with funds contained in an existing money order. The redemption of a money order does not change the money order SMT, it only updates the redeemer's account hash. To prevent double spend (or, more appropriately, double redeem), a money order redemption operation sets the recipient's redemption nonce to the total index of the redeemed money order, i.e. the $4 + 2$ byte position of that money order within the money order batch (2 bytes) within the money order SMT (4 bytes). An account may only redeem money orders with total index is stricly less than their redemption index.

Inbound transfer. This operation inserts tokens from the Ethereum blockchain into the rollup. The transferred funds are locked in a money order that can be redeemed like any other money order. The fromId and blindingFactor of these money orders are set to a default value (e.g. 0).

Outbound transfer. This operation transfers funds from an existing rollup account to an arbitrary Ethereum address by simply burning funds from an account's balance tree and signing the destination Ethereum address.

Every operation type defines an associated rollup state update. To be precise, operators bundle several operations of a given type into standardized batches with an associated proving circuit. Such batches define a single state update. We describe below these kinds of batch transactions. Implementations need not adhere to segregation of operations: it is possible to design circuits that perform any given combination of these operations (within the limits of the proving scheme). One could for instance design circuits performing N_c money order creations and N_r money order redemptions in a single state update, or any other combination of the operations mentioned above. On chain all state update transactions lead only to a state root hash transition. The different transaction types are distinguished through the transaction data.

6.1 Lifetime of the rollup

Our partially anonymous rollup design allows for a limited number of accounts (up to 2^{32}) and a limited number of money order creations (up to $2^{32} \times 2^{16}$, less in practice since money order creation batches need not be full). Thus beyond a certain point no new accounts may be added and no new money orders may be created. *However*, money order redemptions and redemptions can be done even if either (or both) of the account tree / money order tree is full: at no point in time do users lose access to their funds. Indeed, as can be seen from the corresponding proving circuits, money order redemptions and redemptions require only modifying existing accounts, and as such are unaffected by either tree being filled up.

7 Circuits

In this section we go over the zero knowledge proofs associated to the various rollup operations previously discussed. **Convention:** when describing the circuits we underline the $\underline{\textsf{public inputs}}$ of the proof.

7.1 Money order creation

A money order creation operation is the first step in the two steps process for transfering funds from one rollup account to another. As an operation triggered by an account holder it involves two parties: the account holder and its rollup operator. It is finalized on chain with the inclusion of a batch money order creation transaction to the rollup state.

7.1.1 Work of the sender account

The sender account:

1. Creates a plain money order $\textsf{moneyOrder} = [\underline{\textsf{fromId}}, \textsf{toId}, \textsf{tokenId}, \textsf{amount}, \textsf{txBlindingFactor}]$, and computes its hash $\underline{\textsf{moh}}$.

2. Chooses a random scalar s, a random elliptic curve point A and encrypts it with its joint encryption key and with the recipient-operator joint encryption key, thus producing 5 elliptic curve points $[S, B_{\textsf{snd}}, B_{\textsf{snd-op}}, B_{\textsf{rec}}, B_{\textsf{rec-op}}]$.

3. Derives an encryption key κ from A.

4. Uses κ to encrypt (say with a MiMC cipher) the plain money order; specifically it concatenates the following four fields of the plain money order $\underline{\textsf{fromId}}$ (4 bytes), $\textsf{toId}$ (4 bytes), $\textsf{tokenId}$ (2 bytes) and $\textsf{amount}$ (8 bytes) into a single field element

$$\textsf{bufferedMoneyOrder} = \underline{\textsf{fromId}} \cdot 2^{64+16+32} + \textsf{toId} \cdot 2^{64+16} + \textsf{tokenId} \cdot 2^{64} + \textsf{amount}$$

encrypts $\textsf{bufferedMoneyOrder}$ producing a field element $\delta_{\textsf{mo}}$, and encrypts the blinding factor $\textsf{txBlindingFactor}$ producing a field element $\delta_{\textsf{bf}}$,

5. Signs the following 14 field element long message

$$\mathsf{msg} = \big[\underline{\mathsf{moh}},\ \mathsf{txNonce},\ S,\ B_{\mathsf{snd}},\ B_{\mathsf{snd\text{-}op}},\ B_{\mathsf{rec}},\ B_{\mathsf{rec\text{-}op}},\ \delta_{\mathsf{mo}},\ \delta_{\mathsf{bf}},\big]$$

(where $\mathsf{txNonce} = \mathsf{nonce} + 1$ and nonce is $\geq$ sender account's current nonce) against its public key publicKey producing a signature sig.

7.1.2 Money order creation request

The user sends a **money order creation request** to the operator, i.e. the following data:

moneyOrder	a *plain* money order
s	a scalar
A	an elliptic curve point
sig	a signature
txNonce	a transaction nonce

(Note that a user can trigger a money order creation request with any operator; if the user doesn't have their account with the chosen operator they will need to provide a partial opening of their account which the operator can verify against their accountHash.) Note the **transaction nonce** field txNonce. A user may send several (say $n \geq 1$) money order creation requests in quick succession to its operator. If the user's account current nonce is nonce, then the transaction nonces of these money order creation requests must be $\mathsf{txNonce}_1 = \mathsf{nonce} + 1$, $\mathsf{txNonce}_2 = \mathsf{nonce} + 2$, $\dots$, $\mathsf{txNonce}_n = \mathsf{nonce} + n + 1$. This requirement is enforced by the circuit described below. The circuit makes it impossible for the operator the treat money order creation requests out of order, the operator thus needs to order these requests. This also implies that if one of the money order creation requests is missing, then none of the money order creation requests with higher txNonce may be dealt with by the operator until it receives the missing one. In this case the user should create a new money order creation request (it need not use the same parameters s, A, sig).

7.1.3 Transaction vetting by the operator

Upon receiving such a money order creation request, the operator:

1. Verifies that the user's token balance contains at least amount tokens to type tokenId, and retrieves the account's current nonce nonce' and public key publicKey,

2. computes the money order hash moh',

3. computes A from $(S, B_{\mathsf{snd\text{-}op}})$ by Elgamal decryption using its half of the joint encryption key,

4. computes the Elgamal encryption $[S', B'_{\mathsf{snd}}, B'_{\mathsf{snd\text{-}op}}, B'_{\mathsf{rec}}, B'_{\mathsf{rec\text{-}op}}]$ of A using s, the sender-operator and the recipient-operator joint encryption keys,

5. computes κ from A and the encryptions δ'_{mo} and δ'_{bf},

6. assembles the message

$$\mathsf{msg'} = \big[\mathsf{moh'},\ \mathsf{txNonce},\ S',\ B'_{\mathsf{snd}},\ B'_{\mathsf{snd\text{-}op}},\ B'_{\mathsf{rec}},\ B'_{\mathsf{rec\text{-}op}},\ \delta'_{\mathsf{mo}},\ \delta'_{\mathsf{bf}},\big]$$

and verifies that sig is the signature of this message against the user's public key.

7.1.4 Circuit

Having received (and vetted) sufficiently many money order creation requests from its users the operator forms a money order creation batch[5]. It then procedes to a local state update associated to this batch. In the process it produces a zk-proof reflecting the computation.

This computation encompasses in particular:

1. Compute the expectedOldStateRoot from an oldAccountRoot and a oldMoneyOrderRoot and compare it to the oldStateRoot — this legitimizes both these Merkle roots; set currentAccountRoot to be the oldAccountRoot,
2. verify that nCreates (the number of money order creations in the batch) is in range, i.e. $0 <$ nCreates $\leq$ creationBatchCapacity,
3. in a loop of length creationBatchCapacity (the maximum number of money order creations in the batch):
    (a) verify the accountHash against the currentAccountRoot using the associated Merkle proof accountHashMerkleProof; note that this requires finding the bit decomposition of the fromId to route the Merkle proof,
    (b) verify the publicKey and the accountStateHash against the accountHash (this requires the jointEncryptionKey, too),
    (c) open the sender account against the accountStateHash, in particular retrieve the nonce and balanceRoot
    (d) verifiy the relevant tokenBalance against the balanceRoot with the associated tokenBalanceMerkleProof; this requires finding the bit decomposition of the tokenId to route the Merkle proof,
    (e) check that the tokenBalance is sufficient for the transaction (i.e. $0 \leq$ amount $\leq$ tokenBalance),
    (f) update the tokenBalance by subtracting the amount from the current balance,
    (g) update the balanceRoot using the same tokenBalanceMerkleProof
    (h) update the nonce to updatedNonce = nonce + 1,
    (i) verify that the fromId, toId, tokenId, amount from the plain money order are in range,
    (j) compute the expected money order hash expectedMoh and compare it to the moh; the expected money order hash expectedMoh is computed using the fields from the plain money order,
    (k) assemble the msg using the previously vetted moh, the updatedNonce and the cipherText; note that the txNonce field of the money order creation request (which is part of the message to be signed) *is not* part of the proof (not even as a private field); it is replaced here with the updatedNonce; this enforces the correct ordering,
    (l) verify the signature sig of the message msg against the public key,
    (m) compute the expectedUpdatedAccountStateHash and compare it to the updatedAccountStateHash
    (n) compute the updatedAccountHash from the updatedAccountStateHash, the publicKey and the jointEncryptionKey
    (o) update the currentAccountRoot using the updatedAccountHash.

    (when the loop variable excedes nCreates substitute the creation requests with a standardized dummyMoneyOrderCreationRequest; to detect this, introduce a boolean variable that switches to 1 as soon as the stock of real money order creation requests is dealt with and swiches the circuit to one with standardized data and stops updating the currentAccountRoot),
4. compute the moneyOrderBatchRoot of the slice of moh's previously computed,

[5]incomplete batches are possible through padding

5. open the leaf at position moneyOrderBatchId of the money order tree (it contains 0) using a moneyOrderBatchMerkleProof; note that this requires the bit decomposition of moneyOrderBatchId; verify it against the oldMoneyOrderRoot,
6. compute the updatedMoneyOrderRoot using the previous Merkle path and the freshly computed moneyOrderBatchRoot
7. compute the expectedNewStateRoot from the updatedMoneyOrderRoot and the currentAccountRoot and compare it with the newStateRoot.

7.1.5 Public data

The **public data** of this proof is

- oldStateRoot i.e. the current state root,
- newStateRoot i.e. the state root after update,
- nCreates the number of money order creations in the batch,
- for every money order in the batch:
  - the sender ID fromId,
  - the updatedAccountStateHash,
  - the cipherText comprised of $[S, B_{\mathsf{snd}}, B_{\mathsf{snd\text{-}op}}, B_{\mathsf{rec}}, B_{\mathsf{rec\text{-}op}}; \delta_{\mathsf{mo}}, \delta_{\mathsf{bf}}]$,
  - the money order hash moh,
- the moneyOrderBatchId, i.e. the 4 byte index of the leaf of the money order SMT where the money order batch root hash is to be inserted.

Note: one may include the moneyOrderBatchRootHash as public data. This saves operators from computing the money order batch root hash from the list of money order hashes while adding only one item to the multi-exponentiation and a single constraint to the circuit. This considerably simplifies the task of updating the money order SMT.

Note: creationBatchCapacity is technically public, but it is hardcoded into the circuit.

7.1.6 Private data

The **private data** of this proof is

- the account details of the sender accounts,
- the plain money orders,
- all Merkle proofs:
  1. the accountHashMerkleProofs used to check the accountHash's against the ever evolving currentAccountRoot,
  2. the tokenBalanceMerkleProofs used to check the tokenBalance's against the balanceRoot's
  3. the moneyOrderBatchMerkleProof used to check the 0-leaf of the money order tree against the oldMoneyOrderRoot.

7.1.7 On chain transaction

In order to enact the batch inclusion of the money orders the operator needs to send a transaction to the partially anonymous rollup smart contract. This transaction includes

List_fromIds	list of $\underline{\mathsf{fromId}}$'s
List_mohs	list of money order hashes $\underline{\mathsf{moh}}$'s
List_updatedAccountStateHashes	list of $\underline{\mathsf{updatedAccountStateHash}}$'s
List_cipherText	list of $\underline{\mathsf{cipherText}} = [S, B_{\mathsf{snd}}, B_{\mathsf{snd\text{-}op}}, B_{\mathsf{rec}}, B_{\mathsf{rec\text{-}op}}; \delta_{\mathsf{mo}}, \delta_{\mathsf{bf}}]$
$\underline{\mathsf{nCreates}}$	number of created accounts
$\underline{\mathsf{newStateRoot}}$	updated state root hash
π	proof of the state root hash transition

Note that the $\underline{\mathsf{oldStateRoot}}$ and $\underline{\mathsf{moneyOrderBatchId}}$ are already found on chain ($\underline{\mathsf{moneyOrderBatchId}}$ starts at 1 (recall that our design forbids a zero-th money order creation) and gets incremented by 1 with every successful money order creation batch. Furthermore note that $\underline{\mathsf{nCreates}}$ is the common length of all four lists List_fromIds, List_mohs, List_updatedAccountStateHashes and List_cipherText and can thus be deduced from these by the smart contract.

7.2 Recognizing the inclusion of a money order

The issuer of a money order can observe its inclusion in the on chain partially anonymous rollup state as follows. It first searches for those rollup state root hash updates generated by its operator which may contain said money order. It then looks for its account ID in the money order creation transaction data. For every appearance of its ID it compares the associated money order hash with the hash of the money order it wants to verify the inclusion of. If there is a match it knows that the rollup state now contains said money order.

Operators (and users if they choose to) can also surveil the flow of transactions. For money order creation batches, for instance, the operator can, for every transaction, try to Elgamal decrypt the Elgamal encryption $(S, B_{\mathsf{rec\text{-}op}})$ ((S, B_{rec}) in the case of users), retrieve a supposed point A' as $A' := B_{\mathsf{rec\text{-}op}} - \mathsf{sk}_{\mathsf{op}} \cdot S$, compute the associated encryption key κ' and decrypt the pair $\delta_{\mathsf{mo}}, \delta_{\mathsf{bf}}$. They check whether the first decryption yields is well formed (i.e. starts with trailing zeros, then a 4 byte from id, a 4 byte to id, a 2 byte token type and an 8 byte amount). They then check that the fromId coincides with the from Id in List_fromIds at the same index, then they compute the money order hash associated with the previous fields and compare it to the money order hash in List_mohs at the same position. If there is a match, they can inform the user with id equal to the decoded toId of the fact that they have received an outstanding money order they may now retrieve.

Users can do the same and thus not have to trust the honesty of their operator.

7.3 Money order redemption

In a money order redemption, a user retrieves funds that are currently locked in a money order creation batch or an inbound batch (both have the same format, are indistinguishable from the redemption circuit point of view, and are stored in the money order batch SMT).

7.3.1 Work of the redeemer

A user, can learn of the fact that there is an outstanding money order for them either by

1. scanning money creation batches posted on chain and deciphering the money order creation that has them as recipient (i.e. the toId is their id) themselves,
2. or by relying on their operator to inform them and decrypting the money order on their behalf,
3. or by having been contacted directly by the issuer of the money order creation (maybe accompanied by transaction details),

4. or having created an inbound transfer for themselves,

Once reconstructed, a money order can be redeemed by assembling a money order redemption request to pass on to their operator. This requires them to

1. Choose a random scalar s, a random elliptic curve point A and encrypt it with their account's joint encryption key, thus producing 3 elliptic curve points $[S, B_{\mathsf{rdm}}, B_{\mathsf{rdm\text{-}op}}]$,
2. Derive an encryption key κ from A.
3. Uses κ to encrypt (say with a MiMC cipher) the concatenation of the following four fields of the plain money order: fromId (4 bytes), <u>toId</u> (4 bytes), tokenType (2 bytes) and amount (8 bytes). This yields a single field element which the redeemer encrypts it producing a field element δ_{mo} (there is no need for the encrypting the blindingFactor).
4. Compute the concatenation of the moneyOrderTotalIndex (a 6 byte integer) with the updated nonce, i.e. $\mathsf{updatedNonce} = \mathsf{nonce} + 1$ (a 4 byte integer) as a field element and encrypts it with the encryption key κ producing δ_{rdm}
5. Sign the following 8 field element long message

$$\mathsf{msg} = \big[\mathsf{txNonce},\ S,\ B_{\mathsf{rdm}},\ B_{\mathsf{rdm\text{-}op}},\ \delta_{\mathsf{mo}},\ \delta_{\mathsf{rdm}}\big]$$

against the redeemer account's public key, producing a signature sig.

7.3.2 Money order redemption request

The user then sends the corresponding money order redemption request to their operator. This request is comprised of:

mo	the plain money order to be redeemed
s	field element
A	elliptic curve point
moneyOrderTotalIndex	6 byte integer position of the money order in the full money ordre tree
sig	a signature.
txNonce	transaction nonce

As with money order creation requests, the user specifies a transaction nonce in order to allow the operator to order the redemption requests emanating from the user's account. Thus if the user wants to trigger k redemptions in quick succession, it sends k money order redemption requests with transaction nonces $\mathsf{nonce}+1$, $\mathsf{nonce}+2$, …, $\mathsf{nonce}+k$ (where nonce is the account nonce before any money order redemptions).

7.3.3 Transaction vetting by the operator

The operator:

1. retrieves the money order hash that was previously inserted at the user provided moneyOrderTotalIndex, computes the money order hash from the user provided mo and compares it to the one previously retrieved,
2. retrieves account details of the account with id <u>toId</u> from the plain money order, in particular its public key publicKey, its joint encryption key jek, its nonce, redemptionIndex and tokenBalance for the token with Id tokenId from the plain money order,
3. verifies that $\mathsf{redemptionIndex} < \mathsf{moneyOrderTotalIndex}$ holds,
4. computes the encryption of A using s and the joint encryption key thus producing $(S', B'_{\mathsf{rdm}}, B'_{\mathsf{rdm\text{-}op}})$,

5. derives an encryption key κ' from A' and computes the encryptions δ'_{mo} and δ'_{rdm},

6. computes the message

$$\mathsf{msg}' = \left[\mathsf{txNonce},\ S',\ B'_{\mathsf{rdm}},\ B'_{\mathsf{rdm\text{-}op}},\ \delta'_{\mathsf{mo}},\ \delta'_{\mathsf{rdm}}\right]$$

and verifies that sig is the signature of msg' against the publicKey.

7.3.4 Circuit

The circuit retraces the vetting described above. Thus, having received and vetted a number of money order redeption requests from its users, the operator forms a money order redemption batch. It produces to a local state update and in the process produces a zkp for the associated rollup state update. This entails:

1. Computing the expectedOldStateRoot from an oldAccountRoot and a moneyOrderRoot and comparing it to the $\underline{\mathsf{oldStateRoot}}$,

2. Verifying that $\underline{\mathsf{nRedeems}}$ (the number of money order redemptions in the batch) is in range, i.e. $0 < \underline{\mathsf{nRedeems}} \leq \mathsf{redemptionBatchCapacity}$ (the maximum number of money order redemptions in the batch, which is hard-coded into the circuit and technically public),

3. in a loop of length redemptionBatchCapacity :

   (a) compute the money order hash moh from the plain money order

   (b) prove moh's inclusion in the money order tree using a Merkle proof mohMerkleProof leading to moneyOrderRoot (routing requires doing the bit decomposition of the moneyOrderTotalIndex); note that this Merkle proof has length $32 + 16$,

   (c) verify the redeemer's accountHash against the currentAccountRoot using the accountHashMerkleProof: this requires finding the bit decomposition of the plain money order's $\underline{\mathsf{toId}}$ and using it to route the Merkle proof,

   (d) verify the redeemer's publicKey and accountStateHash against the accountHash (this requires the jointEncryptionKey, too),

   (e) open the redeemer account against its accountStateHash, in particular retrieve its nonce, redemptionIndex, the relevant tokenBalance (the tokenBalance is verified against the balanceRoot with a Merkle proof balanceMerkleProof; this requires the bit decomposition of the tokenId from the plain money order to route the corresponding Merkle proof),

   (f) check that $\mathsf{redemptionIndex} < \mathsf{moneyOrderTotalIndex}$,

   (g) check that the $\mathsf{tokenBalance} + \mathsf{amount}$ does not overflow (i.e. is $< 2^{64}$),

   (h) update the tokenBalance by adding the amount from the current balance,

   (i) compute the updated balanceRoot using the same Merkle path that that certified the old tokenBalance

   (j) set the account's nonce to $\mathsf{updatedNonce} = \mathsf{nonce} + 1$, (the account's nonce is used to open the account and to verify the signature, ensuring "for free" that the value of the redemption transaction matches the one of the account).

   (k) set the account's redemptionIndex to the moneyOrderTotalIndex,

   (l) verify the signature sig of the message msg against the public key; as was the case with money order creations, the txNonce isn't part of the proof (not even as a private field), and in the message msg it is replaced with the updatedNonce,

   (m) compute the expectedUpdatedAccountStateHash and compare it to the $\underline{\mathsf{updatedAccountStateHash}}$

   (n) compute the updatedAccountHash from the $\underline{\mathsf{updatedAccountStateHash}}$, the publicKey and the jointEncryptionKey

(o) update the currentAccountRoot using the updatedAccountHash and the same Merkle proof that verified the accountHash previously,

(when the loop variable excedes nRedeems substitute the redemption requests with a standardized dummyMoneyOrderRedemptionRequest; to detect this, introduce a boolean variable that switches from 0 to 1 as soon as the stock of real money order creation requests is dealt with and switches the circuit to one with standardized data and stops updating the currentAccountRoot),

4. compute the expectedNewStateRoot from the moneyOrderRoot and the currentAccountRoot,
5. compare the expectedNewStateRoot with the newStateRoot.

7.3.5 Public data

The **public data** of this proof is

- oldStateRoot i.e. the current state root,
- newStateRoot i.e. the state root after update,
- nRedeems the number of money order creations in the batch,
- for every redemption in the batch:
  - the redeemer Id, i.e. toId in the plain money order being redeemed,
  - the updatedAccountStateHash,
  - the cipherText comprised of $[S, B_{\mathsf{rdm}}, B_{\mathsf{rdm\text{-}op}}; \delta_{\mathsf{mo}}, \delta_{\mathsf{rdm}}]$.

7.3.6 Private data

The **private data** of this proof is

- the account details of the redeemer accounts,
- the plain money orders being redeemed and the associated moneyOrderTotalIndex's,
- the moneyOrderRoot,
- the Merkle proofs needed to verify
  1. mohMerkleProof verifying the money order hash inclusion against the moneyOrderRoot,
  2. accountHashMerkleProof verifying the accountHash against the ever evolving currentAccountRoot,
  3. balanceMerkleProof verifying the tokenBalance against the balanceRoot.

7.3.7 On chain transaction

In order to enact money order redemption batch, the operator needs to send a transaction to the partially anonymous rollup smart contract. This transaction includes

List_redeemerIds	list of redeemer ids, i.e. the toId's
List_updatedAccountStateHashes	list of updatedAccountStateHash's, one for every redeemer id
List_cipherText	list of cipherText = $[S, B_{\mathsf{rdm}}, B_{\mathsf{rdm\text{-}op}}; \delta_{\mathsf{mo}}, \delta_{\mathsf{rdm}}]$
nRedeems	common length of the three lists, equal to the number of redemptions
newStateRoot	updated state root hash
π	proof of the state root hash transition

As was the case with money order creation batches, the oldStateRoot is available on chain. Note that nRedeems is the common length of all three lists List_redeemerIds, List_updatedAccountStateHashes and List_cipherText. It can thus be deduced by the smart contract.

7.4 The redemption index

Note that our partially anonymous rollup design does not feature nullifiers for money orders. We therefore require a mechanism to prevent double redeem of money orders. Our solution is to have rollup accounts include a so-called **redemption index**. The purpose of the redemption index is to record the moneyOrderTotalIndex of the last redeemed money order, i.e. the 6-byte index of the money order that was last redeemed by that account (4 bytes for the index of the money order batch in the money order SMT, 2 bytes for the index of the money order hash within the batch). To be valid, money order redemption requests must satisfy that the index of the money order being redeemed be *greater* than the redemption index of the account. The accompanying account update then sets the redemption index to the moneyOrderTotalIndex of the money order that was just redeemed (on top of changing the relevant token balance, balance root, updating the nonce, . . .) Since the account data may be known to the operator used by an account holder we also require the money order redemption request contain a signature of the data, in particular of the redemption index. Otherwise a malicious operator with access to the account data may redeem money orders with a large index thus preventing the account from ever redeeming any outstanding money order with smaller index.

Note that accounts start life with a redemption index set to 0: therefore, no account may ever redeem the money order with index 0 (i.e. the first money order in the first money order batch.) To simplify things we assume that the first money order batch comes pre-filled.

7.5 Account creation

7.5.1 Preliminary work to account creation

A user wishing to create an account must select an operator to open their account with, generate some private fields and inform the operator of relevant account details. Certain fields of the account don't need to be specified (nonce, $\mathsf{redemptionIndex}$ and all balances are initially set to 0), others are communicated to the operator and no one else ($\mathsf{blindingFactor}$) and others still are public ($\underline{\mathsf{publicKey}}$ and $\underline{\mathsf{jointEncryptionKey}}$).

The user thus

1. generates a $\mathsf{privateKey}$ and the associated $\underline{\mathsf{publicKey}}$,
2. queries the operator's public key M_{op} common to all joint encryption keys it has with its users, generates a secret key sk_{u} and associated public key $M_{\mathsf{u}} = \mathsf{sk}_{\mathsf{u}} \cdot g$; this produces the following joint encryption key $(M_{\mathsf{u}}, M_{\mathsf{op}})$,
3. generates a random $\mathsf{blindingFactor}$.

It then needs to communicate parts of these informations to the operator. This is done as follows, the operator:

1. samples a random elliptic curve point A and a random scalar s and encodes A using the joint encryption key, thus producing 3 elliptic curve points $[S, B_{\mathsf{u}}, B_{\mathsf{op}}]$
2. derives an encryption key κ from A and encodes its account's $\mathsf{blindingFactor}$ (e.g. using a MiMC cipher), thus producing δ_{bf},
3. computes the signature sig of the following 14 field element long message

$$\mathsf{msg} = \big[\mathsf{blindingFactor}, \underline{\mathsf{publicKey}}, \underline{\mathsf{jointEncryptionKey}}, S, B_{\mathsf{u}}, B_{\mathsf{op}}, \delta_{\mathsf{bf}}\big]$$

against its $\underline{\mathsf{publicKey}}$.

7.5.2 Account creation request

The user then assembles an **account creation request** comprised of the following fields:

s	a scalar
A	an elliptic curve point
$\mathsf{blindingFactor}$	the blinding factor
$\underline{\mathsf{publicKey}}$	the public key
M_{u}	the user's portion of the joint encryption key
sig	a signature agains the account's public key

and sends it to the chosen operator.

7.5.3 Transaction vetting by the operator

The operator

1. assembles the joint encryption key from M_{u} and its own M_{op},
2. uses s and A to construct the associated encryption of A using the joint encryption key, yielding $(S', B'_{\mathsf{u}}, B'_{\mathsf{op}})$
3. derives an encryption key κ' from A and encodes the $\mathsf{blindingFactor}$ yielding δ'_{bf},
4. assembles the message

$$\mathsf{msg}' = \big[\mathsf{blindingFactor}, \underline{\mathsf{publicKey}}, \underline{\mathsf{jointEncryptionKey}'}, S', B'_{\mathsf{u}}, B'_{\mathsf{op}}, \delta'_{\mathsf{bf}}\big]$$

and checks that sig is the signature of msg' against the public key $\underline{\mathsf{publicKey}}$.

7.5.4 Circuit

The operator forms a money order creation batch from vetted account creation requests (padding the batch if necessary using a standardized default account creation request). It procedes to a local state update associated to this batch. In the process it produces a zk-proof reflecting the computation.

This computation encompasses in particular:

1. Compute the $\mathsf{expectedOldStateRoot}$ from an $\mathsf{oldAccountRoot}$ and a $\mathsf{moneyOrderRoot}$ and compare it to the $\underline{\mathsf{oldStateRoot}}$ — this legitimizes both these Merkle roots; set $\mathsf{currentAccountRoot}$ to be the $\mathsf{oldAccountRoot}$,
2. verify that $\underline{\mathsf{nAccountCreations}}$ (the number of real account creations in the batch) is in range, i.e. $0 < \underline{\mathsf{nAccountCreations}} \leq \mathsf{accountCreationBatchCapacity}$ and that the batch won't overflow the account tree (i.e. $\underline{\mathsf{nAccountCreations}} + \underline{\mathsf{maxId}} < 2^{32}$, where $\underline{\mathsf{maxId}}$ is the greatest $\mathsf{accountId}$ that's already allocated)
3. in a loop over the variable $0 < i \leq \mathsf{accountCreationBatchCapacity}$ (the maximum number of money order creations in the batch):
    (a) set $\mathsf{accountId} = \underline{\mathsf{maxId}} + i$ and verify the $\mathsf{emptyAccountMerkleProof}$ linking the 0 leaf at position $\mathsf{accountId}$ to the $\mathsf{currentAccountRoot}$ (and routed by means of the bit decomposition of $\mathsf{accountId}$),
    (b) compute the $\mathsf{expectedAccountStateHash}$ using $\mathsf{nonce} = 0$, $\mathsf{redemptionIndex} = 0$, the $\mathsf{blindingFactor}$ and $\mathsf{balanceRoot} = \underline{\mathsf{emptyBalanceRoot}}$ where $\underline{\mathsf{emptyBalanceRoot}}$ is a public value equal to the Merkle root of a binary Merkle tree of depth 16 with all its leaves equal to 0; note that this could instead be recomputed in circuit with every batch (16 hashes) or be hardcoded into it,
    (c) compare the $\mathsf{expectedAccountStateHash}$ to the $\underline{\mathsf{accountStateHash}}$,

(d) compute the accountHash from the publicKey, the jointEncryptionKey and the expectedAccountStateHash,

(e) compare the expectedAccountStateHash with the accountStateHash

(f) verify the signature sig of the message

$$\mathsf{msg} = \left[\mathsf{blindingFactor}, \underline{\mathsf{publicKey}}, \underline{\mathsf{jointEncryptionKey}}, S, B_{\mathsf{u}}, B_{\mathsf{op}}, \delta_{\mathsf{bf}}\right]$$

against the public key publicKey,

(g) update the currentAccountRoot using the accountHash and the same Merkle proof as was used to open the 0 leaf previously

(when the loop variable excedes nAccountCreations substitute the creation requests with a standardized dummyAccountCreationRequest; to detect the overflow, introduce a boolean variable that switches to 1 as soon as the stock of real money order creation requests is dealt with and swiches the circuit to one with standardized data and stops updating the currentAccountRoot),

4. compute the expectedNewStateRoot from the moneyOrderRoot and the currentAccountRoot,

5. compare the expectedNewStateRoot with the newStateRoot.

7.5.5 Public data

The **public data** of this proof is

- oldStateRoot i.e. the current state root,
- newStateRoot i.e. the state root after update,
- nAccountCreations the number of account creations in the batch,
- for every account creation in the batch:
  - the accountStateHash,
  - the jointEncryptionKey and publicKey,
  - the cipherText comprised of $[S, B_{\mathsf{u}}, B_{\mathsf{op}};, \delta_{\mathsf{bf}}]$,
- the maxId before the account creations,
- the emptyBalanceRoot (Merkle root of a binary Merkle tree of depth 16 with all leaves = 0.)

7.5.6 Private data

- All Merkle proofs emptyAccountMerkleProof linking empty accounts to the ever evolving currentAccountRoot,
- the moneyOrderRoot and oldAccountRoot and newAccountRoot
- all the blindingFactor's.

7.5.7 On chain transaction

The partially anonymous rollup smart contract requires the following transaction data for an account creation batch:

List_accountStateHashes	list of accountStateHash's of the newly created accounts
List_cipherText	list of cipher texts $[S, B_{\mathsf{u}}, B_{\mathsf{op}};, \delta_{\mathsf{bf}}]$
nAccountCreations	number of created accounts
newStateRoot	updated state root hash
π	proof of the state root hash transition

The oldStateRoot and maxId are already found on chain. Note that nAccountCreations is the common length of both List_accountStateHashes and List_cipherText and as such can be deduced by the smart contract.

7.6 Inbound transfers

7.6.1 Work of the sender

An inbound transfer has a user send funds directly to the rollup smart contract. The transfer is accompanied by a recipient id toId required to form a plain money order. The token amount amount and token type tokenId are deduced from the transfer itself. The sender id and blinding factor required to form a plain money order are given default values: fromId $= 0$ and blindingFactor $= 0$. No encrypted data is necessary.

These fields are to be concatenated into a single field element in the smart contract which we denote by

$$\mathsf{concatenatedInbound} = \mathsf{toId} \cdot 2^{64+16} + \mathsf{tokenId} \cdot 2^{64} + \mathsf{Amount}$$

7.6.2 Circuit

Inbound transfers (i.e. the transfer of funds from an Ethereum account to the rollup) are the means by which funds enter the rollup. Inbound transfers are queued in order or reception in the rollup smart contract. An inbound transfer is *pending* if it exists in the rollup smart contract but hasn't yet been included in the rollup state. Inbound transfers that have been included in the state (via in an Inbound transfer state update) are removed from the queue. Operators assemble inbound transfers in chronological order to form batches. An inbound batch modifies the money order tree by inserting a money order batch in the rollup's money order tree (existing rollup accounts may later perform a money order redeption to retrieve these funds.) The account tree isn't modified.

The circuit thus

1. computes the expectedOldStateRoot from an accountRoot and a oldMoneyOrderRoot and compares it to the oldStateRoot — this legitimizes both Merkle roots;

2. verifies that nInbounds (the number of real inbound transfers in the batch) is in range, i.e. $0 <$ nInbounds $\leq$ inboundBatchCapacity where inboundBatchCapacity $= 2^{16}$,

3. in a loop over the variable $0 < i \leq$ inboundBatchCapacity compute the moh as

$$\mathsf{moh} = \mathsf{Hash}\big(\big[\ 0 \parallel \underline{\mathsf{concatenatedInbound}}\big]\big)$$

   (note that this is coherent with our definition of the hash of a money order.) In case of an incomplete batch (i.e. nInbounds $<$ inboundBatchCapacity) use a standardized money order)

4. compute the Merkle root moneyOrderBatchRoot of the slice of previously computed money order hashes,

5. open the 0 leaf at moneyOrderBatchId in the money order tree using the associated moneyOrderBatch-MerkleProof; note that this requires computing the bit decomposition of the moneyOrderBatchId to route the Merkle proof

6. compute the newMoneyOrderRoot by inserting the moneyOrderBatchRoot into the money order tree and using the same Merkle proof as in the previous step,
7. compute the expectedNewStateRoot from the newMoneyOrderRoot and the accountRoot,
8. compare the expectedNewStateRoot with the newStateRoot.

7.6.3 Public data

Public data for an inbound transfer batch is comprised of

- oldStateRoot
- newStateRoot,
- nInbounds,
- moneyOrderBatchId,
- the slice of concatenatedInbound's,

7.6.4 Private data

Is comprised of the

- oldMoneyOrderRoot, newMoneyOrderRoot and accountRoot,
- the moneyOrderBatchMerkleProof for inserting the money order batch root hash into the money order tree at position moneyOrderBatchId.

7.6.5 On chain transaction

List_concatenatedInbound	list of concatenatedInbound's
nInbounds	= number of inbound transfers
newStateRoot	updated state root hash
π	proof of the state root hash transition

Note: the oldStateRoot and moneyOrderBatchId are already found in the rollup smart contract. Note that nInbounds is the length of List_concatenatedInbound and can be deduced by the smart contract.

7.7 Outbound transfers

7.7.1 Work of the user

Outbound transfers extract funds out of the partially anonymous rollup account and allow them to return on chain. The user must specify the token type, the amount to be extracted and recipient Ethereum address. As usual, a signature against the account's public key is necessary to validate the operation. No encrypted data is necessary.

The user thus must:

1. compute the following field element

$$\mathsf{bufferedFields} = \mathsf{fromId} \cdot 2^{16+32+64} + \mathsf{amount} \cdot 2^{16+32} + \mathsf{txNonce} \cdot 2^{16} + \mathsf{tokenId}$$

   where txNonce is the transaction nonce and should be $\geq$ nonce $+\,1$ where nonce is the account's current nonce,

2. sign the 2 field element message

$$\mathsf{msg} = [\underline{\mathsf{ethereumAddress}}, \underline{\mathsf{bufferedFields}}]$$

   (interpret the 160 bit Ethereum address $\underline{\mathsf{ethereumAddress}}$ as a field element) against the rollup account's public key, thus producing a signature sig.

7.7.2 Circuit

The task of the circuit is the following:

1. compute the expectedOldStateRoot from an oldAccountRoot and a moneyOrderRoot and compare it to the $\underline{\mathsf{oldStateRoot}}$ — this legitimizes both these Merkle roots; set currentAccountRoot to be the oldAccountRoot,
2. verify that $\underline{\mathsf{nOutbounds}}$ (the number of outbound transfers in the batch) is in range, i.e. $0 < \underline{\mathsf{nOutbounds}} \leq \mathsf{outboundBatchCapacity}$,
3. in a loop of length outboundBatchCapacity (the maximum number of outbound transfers in a batch):
   (a) bit decompose the fromId and use the bits to route the accountHashMerkleProof leading from the accountHash to the currentAccountRoot,
   (b) verify the accountStateHash, jointEncryptionKey and publicKey against the accountHash
   (c) verify the nonce and balanceRoot against the accoutStateHash,
   (d) verify the tokenBalance against the balanceRoot using the associated balanceMerkleProof routed using the bit decomposition of v,
   (e) check that the amount to retrieve is in bounds, i.e. $0 \leq \mathsf{amount} \leq \mathsf{tokenBalance}$,
   (f) update the token balance to $\mathsf{updatedTokenBalance} = \mathsf{tokenBalance} - \mathsf{amount}$ and updates the balance root to updatedBalanceRoot using the updatedTokenBalance and the previously used Merkle proof balanceMerkleProof,
   (g) update the account's nonce to $\mathsf{updatedNonce} = \mathsf{nonce} + 1$
   (h) compute expectedBufferedFields as

   $$\mathsf{expectedBufferedFields} = \mathsf{fromId} \cdot 2^{16+32+64} + \mathsf{amount} \cdot 2^{16+32} + \mathsf{updatedNonce} \cdot 2^{16} + \mathsf{tokenId}$$

   and compare it to $\underline{\mathsf{bufferedFields}}$; again, the txNonce isn't part of the proof and is replaced with updatedNonce in the circuit,
   (i) verify that sig is the signature of the message

   $$\mathsf{msg} = [\underline{\mathsf{ethereumAddress}}, \underline{\mathsf{bufferedFields}}]$$

   against the account's publicKey,
   (j) compute the expectedUpdatedAccountStateHash and compares it to the $\underline{\mathsf{updatedAccountStateHash}}$
   (k) compute the updatedAccountHash
   (l) compute the updated currentAccountRoot using accountHashMerkleProof previously used to verify the accountHash,

   (a binary variable allows us to deal with the case where $\underline{\mathsf{nOutbounds}} < \mathsf{outboundBatchCapacity}$ by using a standardized outbound transfer and not updating the currentAccountRoot after that counter hits 1)
4. using currentAccountRoot and moneyOrderRoot compute the expectedNewStateRoot and compare it against the $\underline{\mathsf{newStateRoot}}$.

7.7.3 Public data

The public variables of the proof are

- the oldStateRoot
- the newStateRoot
- nOutbounds,
- for every outbound transfer:
  - the updatedAccountStateHash,
  - the ethereumAddress,
  - the bufferedFields (from which one can gather the fromId, tokenId and amount)

7.7.4 Private data

The private variables of the proof are,

- the moneyOrderRoot and oldAccountRoot
- for every outbound transfer,
  - the balanceMerkleProof for vetting the token amount against the balanceRoot
  - the accountHashMerkleProof for verifying the accountHash against the ever changing currentAccountRoot,
  - the signature sig, the account's publicKey, jointEncryptionKey, tokenBalance

7.7.5 On chain transaction

List_bufferedFields	list of bufferedFields' describing the outbound transfers
List_ethereumAddresses	list of recipient ethereumAddress's
List_updatedAccountStateHashes	list of updatedAccountStateHash's
nOutbounds	= number of outbound transfers
newStateRoot	updated state root hash
π	proof of the state root hash transition

Again, the oldStateRoot already exists on chain. Note that nOutbounds is the common length of the three lists List_bufferedFields, List_ethereumAddresses and List_updatedAccountStateHashes and can thus be deduced from them by the smart contract.

8 Conclusion

In this note we presented a design for **partially anonymous rollups**. Partially anonymous rollups preserve some of the secrecy properties of anonymous rollups without users having to generate their own zk proofs and thus without operators having to generate recursive proofs. Partial anonymity is achieved by making communication transparent between users and their operators and operators knowing what transactions they are performing, but transactions being opaque on-chain.

Upsides of this design include[6] (1) relatively small state, i.e. two depth 32 Merkle trees, which operators can keep in memory at all times and easily update, allowing the operator to handle thousands of transaction per second without impacting performance (2) simpler proving schemes for operators (3) fewer constraints

[6]All comparisons (simpler, fewer, higher, ...) are with respect to fully anonymous rollups as specified in [3]

per user transaction and thus higher transaction throughput (4) simple account management for users: all of the account information besides its secret key can be recovered from encrypted on-chain data (or from their operator) (5) lightweight user experience: to create transactions users who have their account information stored with an operator need only produce a signature against their account's public key (6) transaction details don't leak on chain (7) data availability for all concerned parties (users (both senders and receivers) and operators) (8) encryption makes off-chain communication between parties in a transaction unnecessary (9) redemptions and outbound transfers remain possible after either (or both) the account SMT or the money order SMT are filled up. Some of its **downsides** include (1) the money orders must be redeemed in their creation order for otherwise the unredeemed money orders with smaller moneyOrderTotalIndex can never be redeemed (2) account *activity* leaks on chain in the form of updates to account hashes (3) the operator performing a transaction is privy to full transaction details (4) participants in a transaction learn their counterparty ID.

References

[1] HarryR, Yondon Fu, Philippe Castonguay, BarryWhitehat, Alex Gluchowski (2018), Roll-up and roll-back snark side-chain with around 17000 transactions per second.

[2] Vitalik Buterin (2018), On-chain scaling at potentially 500 transactions per second.

[3] Olivier Bégassat, Alexandre Belling, Nicolas Liochon (2020), Account based Anonymous Rollup.