Partially Anonymous Rollups

Olivier Bégassat, Alexandre Belling, Nicolas Liochon

v1.4, June 2021

Abstract

This note contains the specification of a kind of rollup with anonymity and scalability properties halfway between those of a simple zk-rollup [1, 2] and those of a fully anonymous zk-rollup as specified in [3]. With partially anonymous rollups, the pperators participating in the transaction can access the transaction details before executing it. The protocol we propose here is account based and allows for a very high transaction throughput. 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 and operators officialize/enact state updates on chain by means of updating the rollup's state root hash stored in the rollup smart contract. 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 inform operators about the transactions they wish to do, in particular whether users have to reveal transaction details or not.

Operator/Blockchain Communication: how operators justify state root hash transitions to the rest of the world, 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 state root hash transitions associated to a batch of transaction with a transparent proof of computational integrity. In particular, *full transaction details* are published as part of the transaction data of the blockchain transaction realizing the rollup state root hash update. On chain transparency is required for other operators to be able to update their view of the rollup state. In a fully anonymous rollup, users send their desired transactions to the operator but 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 of a transaction), rather they provide operators with a proof of a valid account transition justified against an account root hash or the rollup state root hash. 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

[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.

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 amounts 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, but operators communicate state updates on chain (and thus to other operators) only through *Hashes*: updated Account *Hashes*, Money Order *Hashes* and updated State Root *Hashes*.

	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 Money Order Hash and claiming of the funds locked therein).

2 State

2.1 The state — operator view

The state of a Partially Anonymous Rollup, as witnessed by an operator, is comprised of the following parts:

1. (in memory) the Account sparse Merkle tree (SMT),
2. (in memory) the Money Order SMT.

Both the Account SMT and the Money Order SMT are 2-ary sparse Merkle trees of depth 32 in append mode. Both are stored in memory. The leaves of both contain 32 Byte integers. The leaves of the Account SMT contain **Account Hashes**, i.e. the hashes of an auxiliary data structure, called an **Account** which we explain below. Operators do not need to know the contents of these data structures (and generally will not). The leaves of the Money Order SMT contain **Money Order Batch Root Hashes**, which we also explain below. Both of these trees are updated continually as updates roll in. This involves inserting leaves in either tree, updating the leaves of the Account SMT and in both cases updating all relevant Merkle paths (in particular updating the root hashes of both trees).

Operators may also store (on disk) certain batches of Money Order Hashes (whose root hashes populate the leaves of the Money Order SMT). To be precise, operators may choose to remember the Money Order Hashes from those Money Order Batches they oversaw the inclusion of: the Merkle paths in those batches are necessary for Money Order Redemption, and their users may request them.

2.2 The state — on-chain view

The smart contract of a Partially Anonymous Rollup contains a single root hash, the "State Root Hash" representing the rollup's state. This root hash is in fact the hash of the Account SMT root hash and the Money Order SMT root hash:

$$\text{State RH} = \mathsf{Hash}\Big[\text{Money Order SMT RH} \,\Big\|\, \text{Account SMT RH}\Big]$$

[2] We explain Money Orders below.

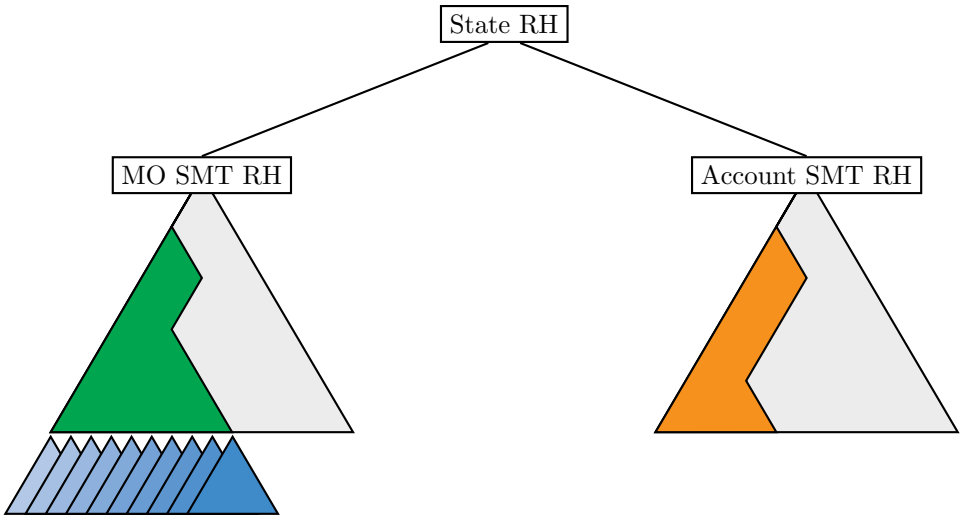

Figure 1: The Money Order SMT and Account SMT. The leaves of the Money Order SMT are the root hashes (RH) of Money Order Batches (represented by blue triangles) which operators n't need to keep in memory but are used implicitely (e.g. as private data in proofs: Merkle paths linking individual Money Orders to a batch root hash.)

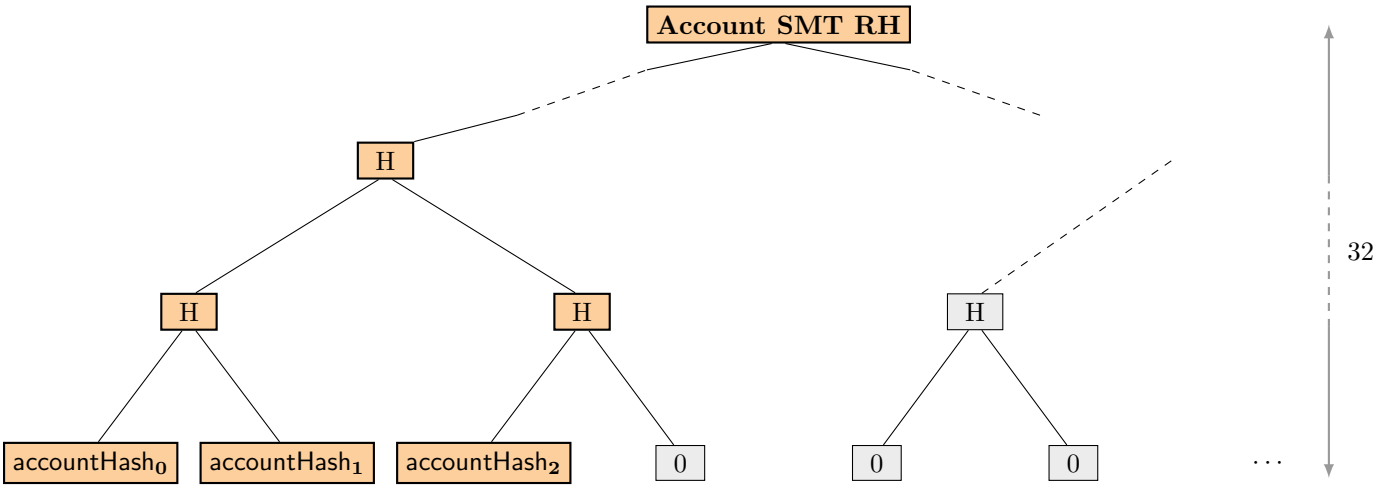

Figure 2: Every nonempty leaf of the Account SMT contains the Account Hash of the Account with ID the position of the leaf in the tree. Here there are 3 existing accounts.

Transactions sent to the Partially Anonymous Rollup smart contract update this root hash, by implicitely updating either the Account SMT or simultaneously both the Account SMT and the Money Order SMT.

3 Accounts and the Account SMT

3.1 Accounts

Accounts are described by the following data:

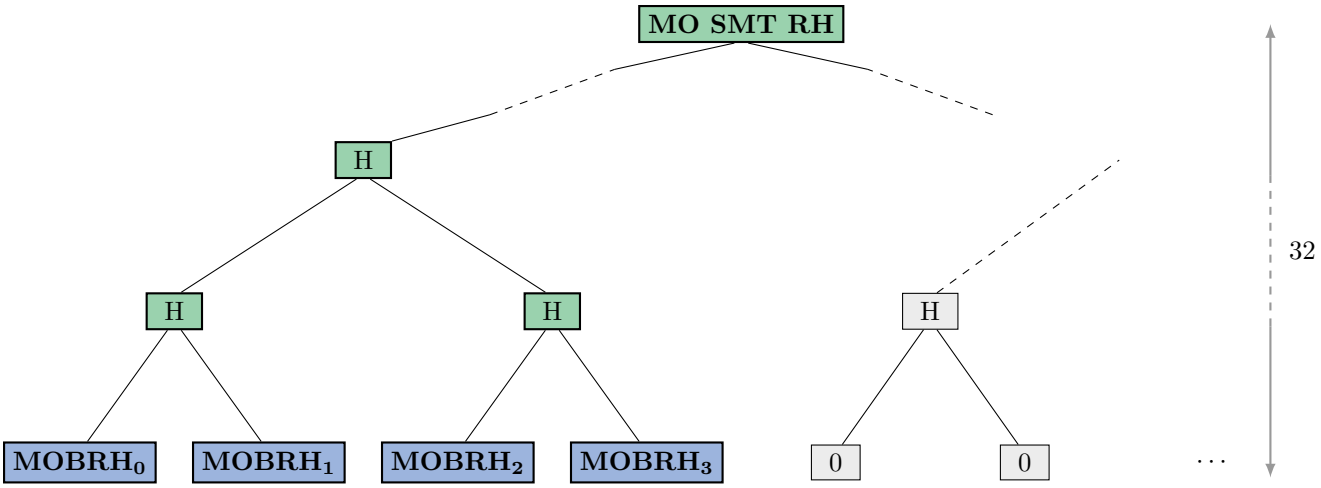

Figure 3: Every nonempty leaf of the Money Order SMT contains a Money Order Batch Root Hash (abbreviated MOBRH, details explained below). In the diagram above the state contains 4 Money Order Batches.

accountId	4 Byte integer
publicKey	Elliptic curve point
nonce	4 Byte integer
redemptionIndex	6 Byte integer
blindingFactor	32 Byte integer
balanceRoot	8 Byte integer
jointEncryptionKey	3 Elliptic curve points

We call accountDetails the collection of the publicKey, nonce, redemptionIndex, blindingFactor and balanceRoot , i.e. an account's accountDetails are all of the fields above *with the exception* of its accountIdand jointEncryptionKey. We call account the accountDetails *plus* the accountId and the jointEncryptionKey. We make this distinction since at times (e.g. opening the account) we need to specify the accountId and at others we don't — it's convenient to have terminology to distinguish between the two.

At account creation the nonce, redemption index and balance are set to 0. The nonce plays the usual role of preventing replay attacks and is incremented with every operation triggered by the account. We explain the redemptionIndex in subsection 7.7. 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 decided by the operator enacting the account creation. It is the account hash's position within 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.

We introduce the term accountDetails all the account fields *except* the accountId. Thus the accountDetails comprise its public key, nonce, redemption index, blinding factor and balance.

3.2 Account Hashes

The Account Hash of an account is the Merkle root of a shallow Merkle tree containing the accountDetails, i.e. all the fields of the account except for the account ID, and the account-operator's Joint Encryption Key. The fields of the account are organized in a tree structure as different account operations require opening different subsets of the account data.

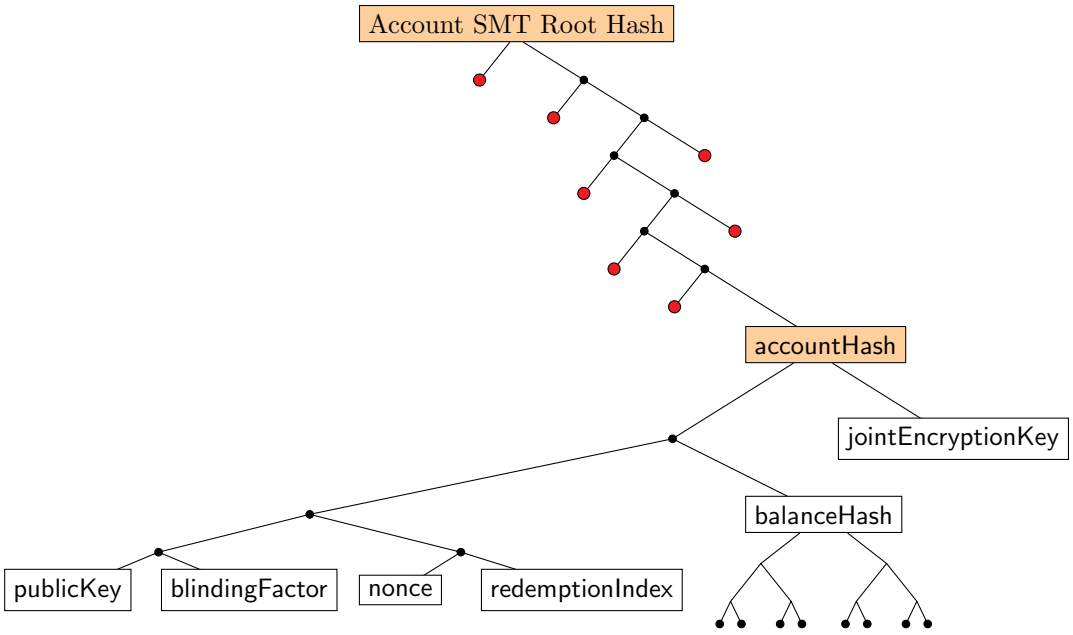

Figure 4: Representation of the internals of an account and the nested structure of the accountHash along with the Merkle Path linking the accountHash to the Account SMT Root Hash.

3.3 The Account sparse Merkle tree (SMT)

The Account SMT is a depth 32 arity-2 SMT whose leaves are indexed by account IDs (ranging from 0 to $2^{32} - 1$). Each leaf contains the Account Hash of the account with that account ID. Leaves with an account ID greater than the current greatest allocated account ID 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 (i.e. batch account creations),
2. updating the account Hashes of existing accounts as the accounts are updated either through Money Order Creation, Money Order Redemption, inbound and outbound transfers,
3. and, for every modification of the leaves, updating its Merkle path and the overall Account SMT Root Hash.

The Account SMT starts out empty (i.e. all its leaves contain 0). Each Batch Account Creation Transaction inserts new Accounts linearly (i.e. the tree is in append mode).

The Account SMT can hold up to $2^{32} \simeq 4.29 \cdot 10^9$ 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 doubles the memory requirements on the operator side and adds a single hash to every Merkle proof involving opening or updating an account.

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

All rollup operations get funneled through an operator who is thus made aware of some or all of the fields of the account. 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 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 Order Creation/Redemption), changing it only when switching operators is sufficient.

3.6 Multi Token Partially Anonymous Rollups

The account structure we describe contains a single balance field. It is possible to replace that balance with a balance root hash, i.e. the root hash of a fixed depth Merkle tree whose leaves are associated with particular tokens. At account creation accounts would start out with the balance root hash set to that of the tree with all leaves containing 0.

4 Encrypted on-chain data and data recovery

4.1 Joint encryption key

Every user-operator pair has an associated **Joint Encryption Key**. The creation of this key uses existing private key / public key pairs of the operator and user: $\mathsf{sk_u} \,/\, \mathsf{pk_u}$ and $\mathsf{sk_{op}} \,/\, \mathsf{pk_{op}}$. These are used to produce a public triple (C_0, C_1, C_2) we call the Joint Encryption Key. These points are defined as follows:

$$\left\{\begin{array}{lcccl} C_0 & = & \omega\,\mathsf{sk_{op}} \cdot \mathsf{pk_u} & = & \mathsf{sk_u}\mathsf{sk_{op}} \cdot H \\ C_1 & = & \omega \cdot (\mathsf{pk_{op}} + \mathsf{pk_u}) & = & (\mathsf{sk_u} + \mathsf{sk_{op}}) \cdot H \\ C_2 & = & \omega \cdot G & = & H \end{array}\right.$$

where G is the chosen base point of the curve, ω is a secret. The operator is free to use different values of ω for all of its users. The triplet is public data associated with a rollup acccount and required in order to issue a Money Order to a given account.

4.2 Encryption of a point with a Joint Encryption Key

Joint Encryption Keys are used to encrypt points M of the curve and establish a shared secret between the issuer, the receiver and their respective operators. Here we explain how to encrypt an elliptic curve point using a Joint Encryption Key. Consider a curve point M and a Joint Encryption Key (C_0, C_1, C_2). To encrypt M one

1. draws a random field element r,
2. computes the triple (M_0, M_1, M_2) defined by

$$\left\{\begin{array}{lcl} M_0 & = & rC_0 \\ M_1 & = & rC_1 \\ M_2 & = & rC_2 + M \end{array}\right.$$

3. computes the hash h of M using a collision resistant hash function.

The tuple (M_0, M_1, M_2, h) is the encryption of M.

4.3 Decryption of a point with a Joint Encryption Key

We describe how to decrypt M from an encryption (M_0, M_1, M_2, h). They key point is that both the operator and the user whose joint encryption key was used in the production of the encryption know a secret value s satisfying $\mathsf{s}^2 \cdot 1 - \mathsf{s} \cdot (\mathsf{sk_u} + \mathsf{sk_{op}}) + 1 \cdot \mathsf{sk_u}\mathsf{sk_{op}} = 0$. which translates to

$$\mathsf{s}^2 \cdot C_0 - \mathsf{s} \cdot C_1 + 1 \cdot C_0 = 0.$$

($\mathsf{s} = \mathsf{sk_u}$ and $\mathsf{s} = \mathsf{sk_{op}}$ are the only solutions to the above system.) Using their respective secret keys the operator and the user may both compute

$$M' = \mathsf{sk}^2 \cdot M_2 - \mathsf{sk} \cdot M_1 + M_0$$

If the triplet (M_0, M_1, M_2) was created using the user-operator Joint Encryption Key, then $M' = M$ and upon computing the hash of M' the operator will find h. Otherwise (since h was computed using a collision resistant hash function) the hashes will be different. This collision proves that the encryption (M_0, M_1, M_2, h) is based on, from the point of view of a user: their joint encryption key with their operator, from the point of view of an operator: one of their joint encryption keys with one of their users.

4.4 Purpose of point encryption with the Joint Encryption Key

The point M is used to derive an encryption key k (e.g. by hashing M using a different hash function from that used to define h). Encrypting M twice, i.e. using the Joint Encryption Key of the recipient (C_0, C_1, C_2) and that of the sender (C'_0, C'_1, C'_2), allows the issuer of the Money Order to forget about the points M it's used historically and to recover them from on chain data. Thus publishing the two encryptions $(M_0, M_1, M_2, M'_0, M'_1, M'_2 h)$ of a single point M allows both the sender and its operator as well as the recipient and its operator to recover the point M purely from on-chain data.

With that shared secret k it is now possible to encrypt and decrypt arbitrary messages that will be readable by the four parties involved.

4.5 Encryption of a Money Order

We apply the previous remark to derive a shared encryption key k from M to encrypt a Money Order on chain in such a way that the issuer of the Money Order and its operator as well as the recipient and its operator can decipher it.

Recall that a Money Order consists of: a 32 bit sender ID, a 32 bit receiver ID, a 64 bit transfer amount and a 256 bit blinding factor for a total of 384 bits of data. We can thus encode a Money Order as two field elements d_1, d_2. Recall the MiMC cipher with key k which leverages a permutation of the base field:

$$\mathsf{MiMC\text{-}Cipher}\big(x; k\big) = F_{90}^k \circ \cdots \circ F_0^k(x)$$

where $F_i^k(y) = (y + c_i + k)^\alpha$ for a small α (depending on the field: $3, 5, 7$ or -1 usually) and round constants $c_0 = 0, c_1, \ldots, c_{89}, c_{90} = 0$.

The MiMC encryption with key k produces two field elements $\delta_1 = \mathsf{MiMC\text{-}Cipher}\big(d_1; k\big)$ and $\delta_2 = \mathsf{MiMC\text{-}Cipher}\big(d_2; k\big)$. Both can be decrypted with knowledge of the key k. Thus all four parties involved can recover the plain Money Order from the encryption (δ_1, δ_2). Note that it's only after decryption of the encrypted Money Order (δ_1, δ_2) that the recipient operator learns which of its users is the recipient: up until that point it only knew one of its user accounts was the recipient.

5 Money Orders and the Money Order SMT

5.1 Money Orders

Money Orders, or **Money Order Base**, are the data structure that describe transactions. A Money Order consists of the transaction details of a transfer of funds between two rollup accounts along with some

random data (the blinding factor) for obfuscation. As such, a Money Order is the data 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
blindingFactor	32 Byte integer	Blinding factor for obfuscation

The fields fromId and toId are account IDs. The tokenId is for multi-token partially anonymous rollups. Any one Money Order concerns *a single token type*. Thus transfering t different token types from one rollup account requires creating t distinct Money Orders. Money Orders contain a field of random data (blindingFactor) for obfuscation.

5.2 Money Order Hashes and Money Ordre Batch Root Hashes

Money Orders are reported on chain in encrypted form: $(M_0, M_1, M_2, M_0', M_1', M_2', h, \delta_1, \delta_2, \mathsf{MOH})$ where (M_0, M_1, M_2, h) and (M_0', M_1', M_2', h) are encryptions of a point M, (δ_1, δ_2) is the encryption of the plain Money Order using the encryption key derived from M and MOH is the Hash of the plain Money Order as described in the previous subsection.

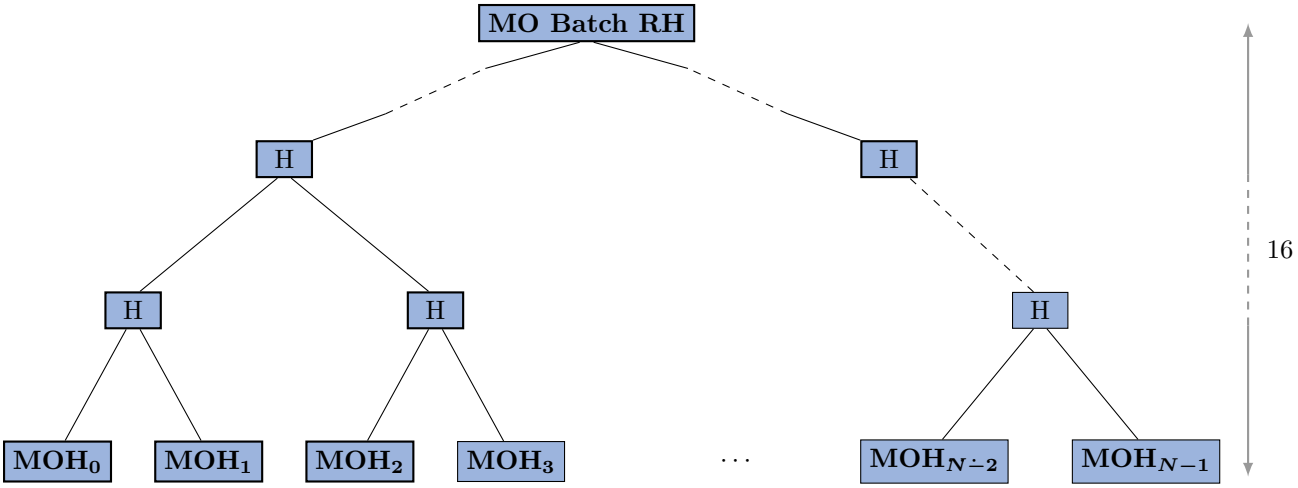

Figure 5: The leaves of a Money Order Batch are up to $N = 2^{16}$ individual Money Order Hashes. The Money Order Batch Root Hash is what actually gets inserted into the Money Order SMT.

Money Orders are included in batches into the Money Order SMT (rather than one by one). **Money Order Batches** contain up to 2^{16} Money Orders per batch. To be precise, the Money Order Hashes MOH (the hash of the fields of a Money Order described in the previous subsection) in a Money Order Batch are ordered by the operator and sequentially inserted in the leaves of a SMT of depth 16. The root hash of this SMT is called the **Money Order Batch Root Hash**. It is this hash which operators insert into the Money Order SMT. 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).

5.3 The Money Order SMT

Along with the Account SMT, the Money Order SMT is an arity-2 depth 32 Sparse Merkle Tree. 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[3] 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. Batches don't have to be full, though, so in practice a Money Order SMT will hold fewer Money Orders than its maximal capacity. Similarly to 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 and doubles the amount of memory needed to store the SMT.

6 Operations

We discuss the various sorts of 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. The newly created account starts life with balance(s) and nonces set to zero and with a public key. The operator computes the Jint Encryption Key and inserts the account hash into the Account SMT. The newly created account receives an account ID (i.e. its position in the Account SMT) upon insertion. The account ID is decided by the operator.

Money Order Creation. Insertion of a Money Order Hash[4] into the Money Order SMT. Such a transaction also updates the Account SMT, namely the Account Hash of every account responsible for issuing a Money Order in the batch.

Money Order Redemption. This kind of transaction credits an account with funds contained in a previously inserted Money Order. The redemption of a Money Order does not change the Money Order SMT. To prevent double spend, a Money Order Redemption operation updates the recipient's Redemption nonce, thus changes the recipient's account Hash and the overall Account SMT.

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.

Outbound Transfer from a Rollup Account. This operation transfers funds from an existing rollup account to an arbitrary Ethereum Address.

Outbound Transfer from a Money Order. This operation transfers funds locked in a Money Order with recipient account B to a an arbitrary Ethereum address. To prevent double spend of the Money Order we require the opening of B's Redemption-nonce, comparing it to the Money Order ID, and incrementing the Redemption-nonce. It also requires the opening of B's public key to verify the signature of the target Ethereum address.

6.1 Associated State updates

Every operation type defines an associated State update. To be precise, operators bundle several operations of a given type into batches that define a single state update. We describe below these kinds of batch transactions. Implementations need not adhere to this strict segregation of operations: it is possible to design circuits that perform a given combination of these operations. One could for instance design circuits

[3] See section 6.

[4] Actually, the Money Order Batch Root Hash of the batch containing it.

performing N_c Money Order Creations and N_r Money Order Redemptions in a single state update, and 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.

7 Proofs

In this section we go over the zk proofs associated to the various rollup operations previously discussed.

7.1 Money Order Creation requests

Every time a user wants to send funds from their rollup account to another rollup account they generate a **Money Order Creation request** and forward it to their operator. Such a request contains:

nonce	nonce
moneyOrder	*plain* Money Order
signature	a signature

(Note that if the user has its Account Data with the operator, it will not need to provide the fields of its account as the operator already knows them. Note on the contrary that if the user doesn't have their Account Data with the operator they will need to provide an opening of their account which the operator verifies against their accountHash; the accountID is used to locate the corresponding accountHash in the Account SMT. The accountID and the account data is used later in the operator generated zkp.)

The account ID of the issuer of the Money Order is contained in the plain Money Order moBase and is necessary for the operator to retrieve the corresponding Account Hash and account data. It is also needed as public data in the proof for routing the Merkle proofs and as transaction data for other operators to know which Account Hashes are to be updated. The account data is needed to establish the legitimacy of the Money Order Creation request (sufficient token balance). It is justified against the current Account Hash of the account ID. The Money Order (Base) moBase contains the details of the transfer (issuer, recipient, transfer amount, blinding factor). The signature is one of the totality of the the provided data:

$$\mathsf{signature} = \mathrm{Sign}(\mathsf{moHash}, \mathsf{nonce}; \mathsf{sk})$$

(where sk is the account's secret key) and is verified against the rollup account's public key pk. It is necessary as a foil against malicious Operators wishing to create transactions using the account's funds. Note that signing the nonce also prevents the operator from enacting valid Money Order Creation Requests out of order. If a user wants to create multiple Money Order Creation Requests in quick succession or at the same time they need to increment the nonce with every signature.

7.2 Encryption

The operator selects a random curve point M. It encrypts the point twice using the jointEncryptionKey (C_0, C_1, C_2) it shares with the issuer of the Money Order and the jointEncryptionKey (C'_0, C'_1, C'_2) of the recipient (which is public data), thus producing $(M_0, M_1, M_2; M'_0, M'_1, M'_2; h)$, where h is a hash of M, as previously described. It also uses the associated encryption key k (a different hash of M to encrypt the plain Money Order thus producing (δ_1, δ_2) as previously described. The operator bundles this data together with the hash of the Money Order MOH.

7.3 Money Order Batch Creation Proofs

We describe in this section how an operator can enact the inclusion of a batch of Money Orders in the Money Order SMT. In broad strokes here is how it happens. The operator receives Money Order Creation requests from users and verifies them. From the set of valid requests it forms a batch of valid Money Order Creation requests and 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:

- Opening the relevant account data in the accounts issuing the Money Orders.
- Verifying that the plain data in the Money Orders is compatible with the account data (sufficient balance, valid signature).
- Retrieving the relevant Joint Encryption Keys from the recipient accounts.
- Computing the encryptions of random curve point M, computing the hashes h and k, and computing the encryptions (δ_1, δ_2) of the plain Money Orders.
- Computing the Money Order Hashes from the plain Money Orders.
- Computing the Updated Account Hashes of the accounts having issued Money Orders (new balance, new nonce, new blinding factor).
- Computing the updated State Root Hash (i.e. updating both the Account SMT and the Money Order SMT).

The goal of the proof is to legitimize a rollup state root (hash) update. The **public data** of this proof is

- for every Money Order in the batch: the account ID of the issuer of the Money Order, the updated Account Hash of the said account, the encryptions $(M_0, M_1, M_2; M_0', M_1', M_2'; h; \delta_1, \delta_2)$, the Money Order Hash of every Money Order,
- the Money Order Batch Root Hash,
- the index of the leaf of the Money Order SMT where the Money Order Batch is to be inserted,
- the *current* State Root Hash as well as the *updated* State Root Hash.

The reason for including the Money Order Batch Root Hash as part of the public data is that 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. This considerably simplifies the task of updating the Money Order SMT.

The **private data** of this proof is

- the account details of the Money Order emitting accounts,
- the points to be encrypted M,
- the plain Money Orders,
- the Merkle Paths needed to verify account hashes against an ever evolving Account SMT root hash.

7.4 Money Order Batch Creation 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

L	list of IDs, encryptions $(M_0, M_1, M_2; M_0', M_1', M_2'; h, \delta_1, \delta_2)$, Money Order Hashes and updated Account Hashes
moBatchRh	Root Hash of Money Order Batch
stateRh	updated State Root Hash
π	proof of the State Root Hash transition

Note that the remaing part of the public data of the proof (the current State Root Hash and the index of the leaf in the Money Order SMT where the MOBRH is inserted) is available inside the rollup smart contract. The proof is verified and if valid, the State Root Hash is updated to the Updated State Root Hash.

7.5 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.

7.6 Money Order Redemption request

With every Money Order Batch rolling in, operators attempt to decrypt the encrypted Money Orders $(M_0, M_1, M_2, M_0', M_1', M_2', h, \delta_1, \delta_2)$. For every encrypted Money Order successfully decrypted it informs the corresponding user of the outstanding Money Order. Users may also run decryption. Once the recipient is in possession of the data related to a Money Order they place a **Money Order Redemption request** with their operator. This includes:

nonce	nonce
moneyOrderIndex	6 Byte integer
mp	Merkle Proof connecting the Money Order Hash to the Batch Root Hash
moneyOrder	the plain Money Order
signature	a signature

The moneyOrderIndex is the 6 Byte integer representing the position of the Money Order being redeemed in the Money Order SMT: 4 Bytes for the Money Order Batch ID of the Money Order Batch containing said Money Order, 2 Bytes for the index of the Money Order in the Batch. (Again, if the redeemer's account data is with its operator, it need not provide account details, in particular it need not provide its redemption-nonce. The signature is, however, mandatory.) The redeeming account ID is contained in the plain Money Order moneyOrder. The signature is against the rollup account's public key and is a signature of the Money Order Hash and the Account Hash:

$$\mathsf{signature} = \mathrm{Sign}(\mathsf{moneyOrderIndex}, \mathsf{nonce}; \mathsf{sk})$$

(Where moneyOrderIndex is the 6 Byte integer specifying the position of the Money Order in the Money Order SMT)Again, if the user needs to place several Money Order Redemption requests in quick succession, they will need to increment the nonce in every successive signature? This prevents out of order redemptions by the operator which would be a problem, see section 7.7. We include the nonce so that the operator may order incoming Money Order Redemption request from a given user.

The operator handling the Money Order Redemption request may not be the same as the operator that included the Money Order into the rollup state, hence providing an opening of the Money Order Hash is necessary. Note that the Merkle path from the Money Order Hash to the Money Order Batch Root Hash can be reconstructed from transaction data. There are four possibilities:

1. The operator that enacted the Money Order Batch Creation sends the relevant Merkle paths "back to its customers", i.e. to the issuers of the Money Orders, who in turn forward it to the recipients.

2. The issuer of the Money Order computes the Merkle path from transaction data and provides the recipient with it along with the opening of the Money Order.

3. The recipient computes the Merkle path on their own using the Batch ID of the Batch containing the Money Order to be redeemed and transaction data.

4. Finally, the Operator carrying out the Money Order Redemption Batch computes the relevant Merkle paths for each redemption request.

The fourth solution is impractical. Operators could in theory compute the full Merkle Trees of each and every Money Order Batch (using the list of Money Order Hashes provided in the transaction data) and store these Merkle Trees in full. However, this would quickly rack up a lot of storage. The better solution is to have redeemers provide their operator with the relevant Merkle Path. Excluding the fourth option, there are three parties that could carry out the computation of these Merkle paths (the operator doing the Money Order Creation Batch, the issuer of the Money Order or its receiver). Since computing these Merkle paths is part of the work an operator is required to do when writing the proof associated to a Money Order Creation Batch, the first solution involves no extra work for the operator besides sending the Merkle paths back to the issuers of the Money Orders. Note that the operator may not know how to contact the receiver account, so sending these Merkle paths back to the issuers is the only sensible option.

Note that operators maintain an up-to-date Money Order SMT and thus are expected to be able to provide the Merkle paths from Money Order Batch Hashes to the current State Root Hash.

7.7 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 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 index of the Money Order that was just redeemed (on top of changing the balance and 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.

7.8 Money Order Redemption Batch Proofs

We describe how an operator can enact a series of Money Order Redemption requests. The operator receives Money Order Redemption requests, tests them for validity and forms a batch of valid Money Order Redemption requests. It procedes to a local state update and in the process writes a proof justifying the transition from the current Rollup State Root Hash to a new Rollup State Root Hash. This proof verifies in particular:

- the openings of the Money Order Hashes to be redeemed (the transfer amount, token type and sender and recipient ID, blinding factor);
- the validity of the Merkle Path from each Money Order Hash to the Money Order Batch Root Hash of the batch containing the Money Order;
- the validity of the Merkle path from this Money Order Batch Root Hash to the Money Order SMT Root Hash (such Merkle paths is what operators are expected to keep in memory);
- the openings of the Account Hash of the recipient IDs, in particular their token balance, their random data, their public key and their Redemption Index;
- that the Redemption Index is less than the index of the Money Order being redeemed,

- the provided signature against the rollup account's public key.

The proof also updates the Rollup State:

- all the recipient accounts (increasing the balance according to the transfer amount in the Money Order, inserting (if necessary) the new blinding factor, setting the Redemption Index to the index of the Money Order being redeemed);
- updating the rollup state root hash.

The **public data** required in this proof is:

1. for every Money Order Redemption, the redeeming ID and its updated Account Hash,
2. the old State Root Hash and the updated State Root Hash.

Again, the account IDs are necessary for routing the Merkle Proofs of the Account Hashes to the state root hash and for updating said Account Hashes. The **private data** is

1. the redeeming accounts' account data, the contents of the Money Orders being redeemed, all Merkle paths, the signatures,

Note that a single account may redeem multiple Money Orders within a single Money Order Redemption batch. When that is the case the account data for every redemption is justified either against the current Account data. Since the Account Data (in particular the Redemption Index) is justified with a signature, this prevents the operator from omitting the inclusion of certain redemptions otherwise treating them out of order.

Finally, note that the Money Order SMT undergoes no update during the Batch Money Order Redemption, only the Account SMT is updated.

7.9 Money Order Batch Redemption Transaction

In order to enact a batch redemption of Money Orders, the operator needs to send a transaction to the Partially Anonymous Rollup Smart Contract. This transaction includes as transaction data

L = [[ID, AccH']]	list of account IDs and updated Account Hashes
State RH'	updated State Root Hash
π	proof of the State Root Hash transition

Note that the remaing part of the public data of the proof (i.e. the current State Root Hash and the index of the leaf in the Money Order SMT where the Money Order Batch Root Hash is to be inserted) is available inside the rollup smart contract. The proof is verified and, if valid, the State Root Hash is updated to the updated State Root Hash.

7.10 Batch Account Creation

Accounts start out with the bare miminum: a public key pk — all other fields (nonce, redemption nonce, balance, blinding factor) set to zero. Account creation requests are sent directly to the Operator of choice. A Batch Account Creation transaction involves a zero knowledge proof consuming a fixed amount of Account Creation requests. Its **public data** is:

1. the first available Account ID (recall that the Account SMT is in append mode) which is available on chain,
2. the current State Root Hash,
3. the updated State Root Hash.

Its **private data** comprises the list of public keys of the accounts being added, the Merkle paths linking the to-be-inserted Account Hashes to the ever changing Rollup State Root Hash. These Merkle paths are known to the operator as operators are required to maintain the up-to-date Account SMT.

Account creations could alternatively be sent to the rollup smart contract. This produces a bottle neck for account creation (when compared to the option of sending these requests directly to operators) but also prevents censorship around account creation and facilitates payment for account creation.

7.11 Inbound Transfers

Inbound transfers (i.e. the transfer of funds from an Ethereum address to the rollup) are the means by which funds enter the rollup and existing rollup accounts can later be credited with external funds. An inbound transfer sends funds directly from an Ethereum address to the rollup smart contract. The funds are accompanied by a *plain* money order, i.e. the datum of

- a recipient ID,
- a token amount deduced from the transfered amount.

Note that we do not specify a blinding factor for an inbound transfer. The details of the Money Order are public so there is no point in specifying a blinding factor. The Money Orders being inserted though an invound transfer still need a blinding factor (in order to be redeemable in the same way as other Money Orders). It seems reasonable to force that blinding factor to be 0 (or some other uniformly enforced value). This enforcement is done in the inbound transfer circuit.

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.

7.12 Inbound Transfer state update proof

An Inbound Transfer rollup state update consists of the insertion of a Money Order Batch Root Hash into the Money Order SMT where all the Money Order (Hashes) in the batch are taken from the list of pending inbound transfers. Inbound Transfers are to be included sequentially in the order the smart contract lists them. Operators are required to include inbound transfers within a number of blocks from their appearance in the rollup smart contract.

Like a Money Order Batch Creation state update, and Inbound Transfer state update inserts a single leaf into the Money Order SMT, it, however, doesn't modify the Account SMT. Operators need to create an inclusion proof which proves a valid State Root Hash transition from the current State Root Hash to that where the Money Order SMT was appended with a single leaf as described above. As such such a proof uses as **public data**:

- the current Rollup State Root Hash,
- the updated Rollup State Root Hash,
- the list of plain Money Orders (in chronological order) to be included and their Money Order Hashes,
- the Money Order Batch Root Hash,
- the Index where to insert the Money Order Batch Root Hash int he Money Order SMT (for routing of the Merkle proofs)

and as **private data**:

- the Merkle Proof from the empty leaf at the Index to the current Rollup State Root Hash.

This Merkle proof is used twice: first to connect the empty leaf to the current State Root Hash and then to connect the (to be inserted Money Order Batch Root Hash) to the updated Rollup State Root Hash.

Note that contrary to the Money Order Creation requests previously discussed, these Money Orders aren't justified against an Account Hash, but against data contained in the rollup Smart contract. The proof involves thus no openings of accounts and no signature verifications.

Money Orders Created through an inbound transfer are redeemed alongside Money Orders created in Money Order Batch Creations: there is only one Money Order Redemption mechanism and it is agnostic as to how the Money Order was inserted into the rollup state.

7.13 Outbound transfer from a rollup account

In this and the next section we discuss how funds exit the Partially Anonymous Rollup and return to an Ethereum address. The first pathway is a transfer from a rollup account to an Ethereum address. This operation requires no Money Orders: the rollup account simply burns funds and signs an Ethereum address. Operators receive outbound transfer requests, check them for validity, create batches of them and batch proofs. The proof for such an operation requires:

- an opening of the Account Hashes,
- comparing of the amount to be transfered back on chain to the account balance,
- verifying a signature of the Hash of the amount to be transfered and the target Ethereum address against the rollup account's public key,
- updating the nonces and balances,
- updating the Account Hashes and the State Root Hashes accordingly.

The **public data** for such a proof is:

- the token amount of each outbound transfer and the target Ethereum addresses,
- the updated Account Root Hashes,
- the updated rollup State Root Hash.

The **private data** are the contents of Accounts being opened, the Merkle paths needed to justify the Account Hashes and the State Root Hashes.

7.14 Outbound transfer from a Money Order

Funds locked in a Money Order can also be funneled directly back to an Ethereum account. The procedure is very similar to the previously discussed outbound transfers. However now we require opening both Money Orders and the recipient Account Hash: indeed, the redemption nonce of the recipient account must be updated to prevent double redeem. This time account holders must provide a signature against their rollup account's public key of the target Ethereum address. Money Orders are redeemed in full so there is to sign the amount to be transfered. Again, operators receive outbound transfer requests from Money Orders, check them for validity, create batches of them and construct batch proofs. The proof for such an operation requires:

- an opening of the Account Hashes (public keys, nonces and redemption nonces),
- an opening of the Money Order Hashes being redeemed back on chain (recipient ID and transfer amount),
- verifying a signature of the Hash the target Ethereum address against the rollup account's public key,

- updating the nonces and redemption nonces,
- updating the Account Hashes and the State Root Hashes accordingly.

The **public data** for such a proof is:

- the target Ethereum addresses,
- the updated Account Root Hashes,
- the updated rollup State Root Hash.

The **private data** are the contents of Accounts being opened, the contents of the Money Orders being opened, the Merkle paths needed to justify the Account Hashes, Money Order Hashes and the State Root Hashes.

7.15 Changing operators

Changing one's operator is simple: it is enough to request one's account information from one's current operator and forward it to a new operator. However, if one were to use the same blindingFactor with the new operator, the previous operator is still in position to guess account details in the future. Indeed, some account details remain the same throughout (e.g. publicKey) others can be reconstructed from transaction data (e.g. the nonce) others an operator may be in position to make educated guesses about or try to brute-force (e.g. the redemptionNonce and balance). Thus switching operators requires more than opening one's account: one should also change one's blindingFactor. This is the purpose of the transaction type we describe here.

7.16 Operator Change Request

An account wishing to switch to a new operator must provide said operator with an **Operator Change Request**. This is a message containing

account	account details and account ID
newBlindingFactor	32 Byte integer, the account's new blinding factor
signature	a signature

The full account opening accountOpening contains all current account details. The operator verifies it against the account Hash associated to the accountId. The signature is that of the newBlindingFactor against the account's public key (which is provided as part of the full account opening.) The message being signed is the newBlindingFactor. Note that the new operator must at this point generate a new Joint Encryption Key for the account. Doing this requires only knowledge of the account's public key.

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[5] (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 per user transaction and thus higher transaction throughput (4) simple account management for users: all

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

of the account information besides its secret key can be recovered from the user's operator (and even from on-chain data, see the appendix) (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. Some of its **downsides** include (1) the money orders must be redeemed in their creation order (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.