(The newest version of this document can always be found on hackmd or GitHub)

We introduce Springrollup: a Layer 2 solution which has the same security assumptions as existing zk-rollups, but uses much less on-chain data. In this rollup, a sender can batch an arbitrary number of transfers to other accounts while only having to post their address as calldata, which is 6 bytes if we want to support up to 2^48 ~ 300 trillion accounts. As a by-product we also achieve increased privacy, since less user data is posted on-chain.

General framework

We start by introducing the general framework that we will use to describe the rollup.

The rollup state is divided in two parts:

- On-chain available state

: State with

on-chain data availability. All changes to this state must be provided as calldata by the operator.

- Off-chain available state

: State without

on-chain data availability. This state will be provided by the operator off-chain.

The on-chain available state can always be reconstructed from the calldata, while the off-chain available state may be withheld by the operater in the worst case scenario (but we will show that our rollup design guarantees that users' funds will still be safe).

The L1 contract stores

- a common merkle state root to both parts of the state.
- the rollup block number
- the inbox

The inbox

is a list of deposit and withdrawal operations that users have added on L1. When posting a rollup block, the operator must process all operations in this list before processing the L2 operations included in the rollup block.

The rollup operator is allowed to make changes to the rollup state by posting a rollup block to the L1 contract, which must include the following as calldata:

1. The new merkle state root.
2. A diff between the old and the new on-chain available state.
3. A zk-proof that there exist a state having the old state root and a list of valid operations (defined below) that when applied to the old state, after processing all operations in the inbox, gives a new state having the new state root, and that the diff provided above is the correct diff.

If the above data is valid, the state root is updated and the inbox is emptied.

Remark:

What we have described so far is a general description of several L2 solutions. For instance:

- If the whole rollup state is in the on-chain available part, and the off-chain available state is empty, we get existing zk-rollups.
- If the whole rollup state is in the off-chain available part and the on-chain available state is empty, we get validiums.
- If both parts of the state contain account state, we get volitions (e.g. zk-porter).

Our proposal is neither of the above, and is described below.

Overview of the rollup design

Transfers

When a user sends L2 transfers to the operator, they are not processed immediately. Instead, they are added to a set of pending transactions in the off-chain available state. After the rollup state has been updated by the operator, the user recieves (off-chain) witnesses to both their balance and to all their pending transactions in the new rollup state from the operator. In order to process their pending transactions, the user signs and sends an operation ProcessTransactions

to the operator. The operator then adds this operation in the next rollup block, which processes all the pending transactions of the sender, and sets a value lastSeenBlockNum(sender) = blockNum

in the on-chain available state, where blockNum

is the last block number. After a rollup block has been posted, the operator provides witnesses to all updated balances to the affected users.

Calldata usage

The only data that needs to be provided as calldata in each rollup block (ignoring deposits and withdrawals) is the set of accounts that have updated their lastSeenBlockNum

, i.e. 6 bytes per address (supporting up to 2^48 ~ 300 trillion accounts). This is already less calldata than regular rollups if each user only added one pending transfer before calling processTransactions

, and is much less per transfer when a user processes a large batch of transfers at once.

Frozen mode

Under normal circumstances, a user may withdraw their funds by sending an L2 transfer to an L1 address that they own. If the transfer is censored by the operator, the user may instead send a ForceWithdrawal

operation to the inbox on L1, which the operator is forced to process in the next rollup block.

If the operator doesn't post a new rollup block within 3 days, anyone can call a Freeze

command in the L1 contract. When the rollup is frozen, users may withdraw the amount determined by

- their balance in a block b

with blockNum >= lastSeenBlockNum(address)

,

- minus

the total amount sent from

the user in the pending transactions in the same block b

(if blockNum == lastSeenBlockNum(address)

),

- plus

the total amount sent to them

in a set of pending transfers in blocks at least as new as b

, that have all been processed.

The user must provide witnesses to all the above data in order to withdraw their funds.

The security of the protocol is proven by showing that each user always has the necessary witnesses to withdraw their funds, which we will do in the detailed description below.

Detailed description of the protocol

Rollup state

Each L2 account's balance is represented as the sum of a balance stored in the on-chain available state and a balance stored in the off-chain available state:

balanceOf(address) = onChainBalanceOf(address) + offChainBalanceOf(address)

The reason for this is to simplify deposits and withdrawals. When a user makes a deposit or a withdrawal on L1, only their on-chain balance is updated. On the other hand, when an L2 transfer is processed, only the off-chain balances of the sender and recipient are updated.

Note that either onChainBalanceOf(address)

or offChainBalanceOf(address)

may be negative, but their sum is always non-negative.

On-chain available state

OnChainAvailableState = { lastSeenBlockNum : Map(L2 Address -> Integer) # The block number of a block in which the owner of the address possess a witness to their balance and pending transactions. , onChainBalanceOf : Map(L2 Address -> Value) # On-chain part of the balance of an account. }

Off-chain available state

OffChainAvailableState = { offChainBalanceOf : Map(L2 Address -> Value) # Off-chain part of the balance of an account. , nonceOf : Map(L2 Address -> Integer) # The current nonce of an account. , pendingTransactions : Set(Transaction) # A set of transactions that have been added, but not processed yet. }

where Transaction

is the type

Transaction = { sender : L2 Address , recipient : L2 address or L1 address , amount : Value , nonce : Integer }

L2 operations

The operator is allowed to include the following operations in a rollup block.

AddTransaction

AddTransaction(transaction : Transaction , signature : Signature of the transaction by the sender)

Adds the transaction to the set pendingTransactions

and increases nonceOf(sender)

by one. It is required that the transaction’s nonce is equal to the current nonceOf(sender)

.

ProcessTransactions

ProcessTransactions(sender : Address , blockNum : Integer , signature : Signature of the message "Process transactions in block blockNum" by the sender)

This operation processes all pending transactions from

sender

in the last published rollup block (i.e. not the currently in-process block), which is required to have block number blockNum

, and sets lastSeenBlockNum(sender)

to blockNum

.

When a transaction is processed, it is removed from pendingTransactions

, the amount is subtracted from offChainBalanceOf(sender)

and added to offChainBalanceOf(recipient)

. If the sender has insufficient funds for the transfer, meaning that amount > balanceOf(sender)

, the transaction fails and is just removed from pendingTransactions

.

The sender should make sure they possess the witnesses for their balance and all their pendingTransactions

in block blockNum

before sending this operation to the operator, since they would need this in order to withdraw in case the rollup is frozen.

L1 operations

The following operations can be added by users to the inbox in the L1 contract.

Deposit

Deposit(toAddress : L2 Address)

Adds the amount of included ETH to onChainBalanceOf(toAddress)

.

ForceWithdrawal

ForceWithdrawal(sender : L2 Address , recipient : L1 Address , signature : Signature of the message "Withdraw all ETH to recipient" by the sender)

Withdraws balanceOf(sender)

ETH to recipient

on L1 and decreases onChainBalanceOf(sender)

by the withdrawn amount (i.e. sets onChainBalanceOf(sender)

to -offChainBalanceOf(sender)

).

Frozen mode

If the operator doesn't publish a new block in 3 days, anyone can call a freeze command in the contract, making the rollup enter a frozen mode

.

When the rollup is frozen, the users that have unprocessed deposits in the inbox can send a call to the L1 contract to claim the deposited ETH in the inbox.

In order to withdraw from an L2 account, a user Alice must provide to the L1 contract the witnesses to the following.

1. offChainBalanceOf(alice)

in some rollup block b

with blockNum >= lastSeenBlockNum(alice)

.

1. If blockNum == lastSeenBlockNum(alice)

, we also require witnesses to the set of pending transactions from

Alice in block b

. We denote the total sent amount as sentAmount

.

1. A set of pending transfers to

Alice. Each pending transfer must have been processed, meaning that it's block cannot be newer than the sender's lastSeenBlockNum

. Also, each pending transfer's block must be at least as new as b

above (otherwise it would already be included in offChainBalanceOf(alice)

). We denote the total recieved amount as recievedAmount

.

When the L1 contract is given the above data, it sends to Alice the amount (if non-negative) given by

offChainBalanceOf(alice) + onChainBalanceOf(alice) + recievedAmount - sentAmount

and decreases onChainBalanceOf(alice)

by the withdrawn amount. If the above amount is negative, the withdrawal request fails and nothing happens.

Remark:

It may happen that Alice withdraws her funds, and then later is made aware of a transfer from Bob that she didn't include in the withdrawal. She may then add a new withdrawal request where she include Bob's transfer along with the same transfers as last time.

Example 1: Single transfer from Alice to Bob

Alice wants to send 5 ETH to Bob. Her current nonce is 7, and her current lastSeenBlockNum

is 67. The procedure is as follows:

1. Alice signs and sends transactiontransaction = (sender = alice , recipient = bob , amount = 5 ETH , nonce = 7)

to the operator.

1. The operator includes the operation AddTransaction(transaction, signature)

in the next rollup block (number 123), with the effect of adding the transaction to the set of pending transactions in the rollup state.

1. After rollup block 123 is published on-chain, the operator sends a witness of the newly added pending transaction to Alice.
2. Once Alice have the witness of her pending transaction in block 123, she signes the message "Process transactions in block 123" and sends this signed message to the operator.
3. The operator includes the operationProcessTransactions(address = alice , blockNum = 123 , signature = Signature of the message "Process transactions in block 123" by Alice)

in the next rollup block, which has block number 124. Alice's lastSeenBlockNum

is set to 123, and the transfer to Bob is processed.

1. The operator gives Alice and Bob the witnesses to their updated balances in block 124.

Security argument

The operator may misbehave in several stages in the example above. If this happens, users can exit by sending a ForceWithdrawal

operation to the L1 inbox. Then, either the operator will process the withdrawal requests in the next rollup block, or it will stop publishing new blocks. If the operator doesn't add a new block in 3 days, anyone can call the freeze command on L1, and the rollup is frozen. For Alice and Bob, there are two scenarios:

- The transfer from Alice to Bob has not been processed (it is either pending or wasn't included at all). Then Alice will use a witness of her balance in some block at least as new as 67 (which is her lastSeenBlockNum

) to exit.

- The transfer was processed, but the operator didn't provide the witnesses to the new balances of Alice and Bob. In this case, Alice have a witness of her balance in block 123 and of the pending transfer to Bob (otherwise she wouldn't sign the ProcessTransactions

operation). Alice can then withdraw using the witness of her balance in block 123, plus a witness to the pending transfer to Bob. Bob may withdraw with a witness to his balance in some block at least as new as his lastSeenBlockNum

, plus a witness of the pending transfer from Alice, which he could get from Alice.

In all both cases, both Alice's and Bob's (and all other user's) funds are safe.

Example 2: Batch of transfers from Alice to 1000 recipients

Suppose Alice is a big employer and want to send salaries to 1000 people. She may then batch the transfers to save calldata. The procedure for this is the same as in Example 1 above, but she will add all 1000 transactions to pendingTransactions

before sending the ProcessTransactions

operation. Note that it is not necessary to add all 1000 transfers in the same rollup block, she may continue to add pending transactions in many rollup blocks before calling ProcessTransactions

.

Discussion

Privacy

This design has increased privacy compared to existing rollups, since an honest operator will not make users balances or transactions public, but only give each user the witnesses to their updated balances.

Token support

We described a MVP without token support, but it is trivial to add support for ERC-20 tokens and NFTs by adding separate balances for these.

Smart contracts

Further research should be done to figure out how to support smart contracts in this design.

Related ideas

- Minimal fully generalized S*ARK-based plasma
- Plasma snapp - fully verified plasma chain
- Plasma snapp-1-bit
- MVR - Minimally Viable Rollback
- Adamantium - Power Users
- A zkRollup with no transaction history data to enable secret smart contract execution with calldata efficiency - #19 by leohio