Cross ZK-Rollup Transfer Protocol

We propose a protocol enabling transfers between two different users

on two different zk-rollups

. The protocol requires an intermediary user which holds an account on both zk-rollups, no trust is required between the users nor the operators. In our proposal, all rollups live on the same smart-contract.

Zk-rollup: link

On-chain scaling to potentially ~500 tx/sec through mass tx validation[

Applications

](/c/applications)

We can actually scale asset transfer transactions on ethereum by a huge amount, without using layer 2's that introduce liveness assumptions (eg. channels, plasma), by using ZK-SNARKs to mass-validate transactions. Here is how we do it. There are two classes of user: (i) transactor, and (ii) relayer. A relayer takes a set of operations from transactors, and combines them all into a transaction and makes a ZK-SNARK to prove the validity, and publishes the ZK-SNARK and the transaction data in a hi…

I Overview

Users

- U1 is on zk-rollup R1 with account R1:ACC1.
- Intermediary user on zk-rollup R1 and R2 with account R1:ACC2, R2:ACC2.
- U3 is on zk-rollup R2 with account R2:ACC3.

Scenario

U1

sends X

tokens to U2

via an intermediary

. The intermediary is not known in advance and it can be any user with an account on both zk-rollup and who is willing to participate in the transaction. The intermediary will receive X tokens from U1

on R1 and pay X

tokens on R2 to U2

.

The total balance of all tokens on each zk-rollup remains unchanged. Users can of course agree between themselves on the transfers to be performed but this would require that they trust each other and the operators

.

II Three phases cross zk-rollup transfer protocol:

For the system to be trustless and secure against Byzantine actors we need following mechanisms:

Locking

U1 locks X tokens on R1 which generates a pending transfer

that is stored on R1.

Finalization

The intermediary transfers X tokens to U2 on R2. This part is safe against double finalization

: ie the protocol guarantees that no more than 1 intermediary can finalize a pending transfer.

Reimbursement

Intermediary is reimbursed by withdrawing the locked funds (from step 1). Only the intermediary who finalized

the pending transfer can unlock and receive the funds.

III ZK-rollup State:

AccountTree

AccountTree is a standard Merkle tree containing all accounts. In addition to the standard vanilla rollup fields', accounts leaves contain the pending transfers and a lockedAmount field.

Account { index, // Index of the public key of the account owner. balance, nonce, lockedAmount, // Amount of locked funds for pending cross-rollups transfers pendingTransferRef, // Hash of the pending cross-rollups transfers for which this account is the sender }

CrossRollupTransfer { fromPKIndex, * toPKIndex, * amount, // Amount to be transfered nonce, // Nonce of the emitter sig, // Signature of {from, to, amount, nonce, rollup-id} receiverRollupId, // Id of the rollup hosting the recipient of the transfer }

pendingTransferRef

which has the hash of CrossRollupTransfer

is enough for the operator to retrieve CrossRollupTransfer data.

FinalizingTransfersTree

FinalizingTransferTree is a sparse Merkle tree containing traces of transfer finalization. We use a sparse Merkle tree to enforce that there cannot be two finalizations for the same cross-rollup transfer using proofs of non-membership by setting positionInTheTree

= pendingTransferRef

(32bytes hash).

FinalizingTransfer { intermediaryIndex, // Index of the pubkey of the intermediary performing the finalization *accountToReimburseIndex, // index of the pubkey to be reimbursed* toIndex, // recipient of the cross rollup transfer* amount, // Amount is the number of tokens delivered by the intermediary to the cross-rollup recipient nonce, // Nonce of the emitter sig, // Signature of {from, to, amount, nonce, rollup-id} pendingTransferRef, // PendingTransferRef = hash(CrossRollupTransfer) }

*As explained in Vitalik's post we maintain a small Merkle tree of public keys by indexes in order to optimize the cost of calldata. (see On-chain scaling to potentially ~500 tx/sec through mass tx validation)

As in vanilla rollup's, we store all the transfers (cross-rollup, finalization, normal-internal-rollup) on chain in the calldata

. Nonces and signatures are not stored in the calldata nor in the transfers Merkle trees because they already in the private inputs of the ZKP.

On-chain State

Only the AccountsTree Merkle root and FinalisedTransferTree Merkle root are kept in smart contract storage.

IV Our solution

1) Locking:

For cross rollup transfers the transferred amount needs to be locked on the sender account and pendingTransferRef

needs to be set to the hash of the cross rollup transfer. This is also checked in the ZKP proof.

from == PK[Index] pendingTransferRef == Hash(from, to, amount, rollupId, nonce) from == EcRecover(TransferReference, Sig) hash{ from, oldBalance, nonce - 1, 0, _ } == AccountTree[Index] amount >= 0 balance - amount >= 0 hash{ from, balance - amount, nonce, locked: amount, pendingTransferRef } = AccounTree[Index]

After the locking step, U1 can’t access locked tokens anymore. We do not describe cancellation workflow in this post.

2) Finalization

After the locking step, the Intermediary decides to complete the transfer on R2. Intermediary sends X tokens to R2:ACC3 and a finalizatingTransfer

is inserted in the sparse Merkle tree at position pendingTransferRef

. We also require that finalizingTransfersTree

[pendingTransferRef

] == 0 before the finalization (ie: the transfer has not been completed before).

The ZKP comprises standard vanilla transfer checks (signature, nonce, amounts, balances etc…) plus:

finalizingTransfersTree[pendingTransferRef] == 0 (before finalization) finalizingTransfersTree[pendingTransferRef] == hash{intermediaryIndex, accountToReimburseIndex, toIndex, amount} (after finalization) pendingTransferRef == Hash(U1, U2, amount, rollupId, nonce)

3) Reimbursement

At this point U1 spent X token on R1 and U2 received X tokens on R2, but the intermediary still needs to claim the X locked tokens from R1 to be even. Intermediary has to prove on R1

that he finalized the pending transfer on R2

.

In the smart contract storage we keep all the Merkle roots of the finalizingTransfersTree

for all batches. Since R1 and R2 are plugged on the blockchain with the same smart-contract, they can access each other’s previous stateMerkleRoots for cheap.

Every update to the finalizedTransfers tree invalidates all the previously issued Merkle proofs. But since finalizingTransferTree is insert-only (and immutable), a proof of membership always remains convincing

even if its Merkle proof (rootHash + path) has been outdated. That’s why we maintain the history of the roots of the finalizedTransferTree on the smart-contract.

The submitted ZKP includes checks that:

hash {intermediaryIndex, accountToReimburseIndex, toIndex, amount} == finalizingTransfersTree[pendingTransferRef] {, , *locked, pendingTransferRef } == accountTree[someIndex] //check the existence of an account with pendingTransferRef Accounts[accountToReimburseIndex].balance += locked* {, , , 0, 0 } == accountTree[someIndex] //Remove the pending transfer

The public inputs are:

- The Merkle roots of AccountTrees before and after the reimbursement of R1
- Some FinalizingTransfer root that includes the finalization we are reimbursing
- pendingTransferRef

V Costs

We consider that the cost in constraint for zk-rollup to be largely dominated by hashes (Merkle proofs), positivity checks and signature verifications. We furthermore assume that the sparse Merkle trees are 80 level deep. Indeed, collisions in the finalizingTransferTree results at worst

in the impossibility to finalize a legitimate cross-rollup transfer.

The account tree is 24 level deep

. We assume that hashing two field elements together has the same cost of hashing each of them separately.

With those assumptions, we get the following (rough estimates) for the number of constraints needed to build the ZKP circuits:

- Locking (~160 Hashes + 1 Signature + 2 PositivityCheck)
- Finalization(~580 Hashes + Signature + 2 PositivityCheck)
- Reimbursement(~280 Hashes)

We estimate the number of hashes for a vanilla transfer to equal 300 Hashes + 1 signature + range check

. The absolute minimum to perform an intermediate type cross-rollup transfer amounts to 2 vanilla transfers : all users trust each other and one transfer is issued on each rollup. Our protocol is only +70%

more expensive than the trusted case.

The smart contract cost is dominated by snark verification and calldata, we store one Merkle root of finalizingTransfersTree per batch in storage but this cost can be neglected. Also, we use Groth16. From those assumptions, we derive a rough estimate of the total gas cost of 600k + calldata costs

.

VI Future improvements

- Support cancellation of cross-rollup transfer. For that, we need a guarantee that the pending transfer has not been completed.
- Multiple cross-rollup transfer at the same time for an account

@AlexandreBelling, @bkolad, @liochon