Celestia Snark Accounts

This is a design document on how Celestia might incorporate SNARK accounts into their base layer for enabling interoperability of TIA between Celestia and the rollups that utilize Celestia for data availability.

The core problem that leads to multiple proposals in this document, is that the SNARK account contract needs a form of replay protection, to prevent double withdrawing. This will lead to a few design choices that we explore below.

Background: Current Transaction Lifecycle on Celestia

In this section, we outline how a regular Celestia account and transaction lifecycle work for context before extending the design to SNARK accounts. Note that the code in this section in pseudocode and more intended as a “philosphical” overview rather than a 1-1 match with the Celestia codebase.

Account State

A account on the Celestia network has the following state:

struct Account { balance: u32 pubkey: Pubkey }

Transaction Lifecycle

A current “send” transaction on Celestia has the following fields:

struct Transaction { signature: Signature sender: Pubkey recipient: Pubkey amount: u32 }

To process this transaction, the following steps are executed (not including gas accounting):

fn process_transaction(tx: Transaction) { verify_signature(tx.signature, tx.sender) assert(balanceOf(tx.sender) >= tx.amount) tx.sender.balance -= tx.amount tx.recipient.balance += tx.amount }

Celestia SNARK Accounts

In this design, each rollup on Celestia has a designated SNARK account that maintains the balance of TIA on the rollup. Users can deposit TIA on the rollup by sending money to the SNARK account (i.e. bridging to the rollup). Users can withdraw TIA from the rollup (withdrawing from the rollup) by providing a proof to the SNARK account that there was a valid withdrawal on the rollup corresponding to their claimed withdrawal on Celestia.

Deposit Transaction: From a Celestia Account to a Rollup SNARK Account

For users wanting to “deposit” money to a rollup, the deposit transaction works like any normal “send” on Celestia. The main difference is that this transaction has an additional “memory” field that can be used to specify additional information (like a recipient address on the rollup):

struct DepositTransaction { signature: Signature sender: Pubkey recipient: Pubkey // the SNARK account of the rollup amount: u32 memo: bytes // memo field that can be used for arbitrary extra information }

- The memo

field should be bytes

to allow for additional extra information tied to a deposit. An example might be that a rollup has a address/public key scheme that might not be the same number of bytes as Celestia’s public key (true for EVM) that can be specified in this memo field.

- When the transaction is processed on Celestia, the balance in the SNARK account address goes up by amount

, and the SNARK account should emit an event with Deposit(sender, amount, rollup_recipient)

- For sequenced rollups, the sequencer should monitor for Deposit

events on their corresponding SNARK account, and then insert “system transactions” (i.e. transactions that only the sequencer can insert) that mint the corresponding amount of TIA to the recipient account on the rollup (can be specified in the memo field).

- For based rollups, deposit transactions to the SNARK account are simply another type of transaction that are inputs to the STF that update the recipient account on the rollup with the deposit amount.

Withdrawal Transaction: From Rollup to the SNARK Account → Recipient account on Celestia

The withdrawal transaction from a rollup to Celestia involves 2 steps:

1. A withdraw transaction on the rollup (to burn TIA tokens on the rollup to mark them as “withdrawn”)
2. Note that whether it is a sequenced or based rollup, withdrawing requires a signed user transaction on the rollup to mark the tokens as “withdrawn”
3. Note that whether it is a sequenced or based rollup, withdrawing requires a signed user transaction on the rollup to mark the tokens as “withdrawn”
4. Transaction on Celestia that prompts SNARK account to send withdrawn tokens to the withdrawal address after proper verification (and also keeps track of an accumulator to prevent double-spend).

Note that there are some designs where the above 2 steps can be combined, but they impose other tradeoffs that we discuss below. We start with assuming there’s a 2 step process for simplicity.

1. Withdrawal transaction on the rollup

To begin the withdrawal, the user sends a transaction to burn the tokens on the rollup (or otherwise mark them as “withdrawn”). This burn transaction includes:

- The amount to be burnt as withdraw_amount
- Address of the recipient account on Celestia as celestia_recipient

(can be an arbitrary memo field as well)

2. Proof of STF to the SNARK account

There are several options with different tradeoffs for how the SNARK account can utilize a proof of the rollup’s STF and then process a user withdrawal on the rollup.

This has a design tradeoff because we must have a solution to prevent “double spends” for withdraws. We briefly sketch high level descriptions of the perceived options, before going into more elaborate detail.

Design decisions which potentially require the base layer to worst-case store state linear in # rollup blocks have been excluded from the tradeoff space.

- Option 1: Withdrawal accumulator updated one at a time
- Option 2: Require-ordered withdrawals, that must process all withdraws in a block range
- Option 3: Blob submission requires proof to update state root
- Option 4: The base layer provides replay protection for recent-history, user proves long-term

Name

Pros

Cons

Withdrawal accumulator updated one at a time

- Similar to Ethereum-style rollups
- Without working with the block builder, only 1 multi-withdraw tx possible per block.

Ordered Withdrawals

- Ordered withdraws maybe a design choice
- Adversarial delay concerns with spamming either Base Layer or Rollup
- Ordered withdraws forced by the base layer, rather than rollup logic

Proof at blob submission time

- Potential less overhead in what blobs need to be processed in ZKP of STF
- Can only submit DA once proof is generated, dramatically increases rollup finality time

Base layer replay protection

- Arbitrarily many withdraws per rollup can occur concurrently per block
- Base layer maintains state for the last week of withdraw txs

Option 1: Withdrawal Accumulator updated one at a time

In this option, the SNARK account keeps track of a “state root” of the rollup and processes “State Update Transactions” that update the state root. There are separate “withdrawal transactions” that users submit that validate that withdrawals against this state root and update the withdrawal accumulator to prevent double-spend.

This option is most similar to how Ethereum-style rollups today do withdrawals from Ethereum. Note that this design works for both based and sequenced rollups.

Snark Account State

struct SnarkAccount { balance: u32 vkey_stf: VerifyingKey vkey_withdrawal: VerifyingKey last_height: u32 state_root: bytes32 withdrawal_accumulator: bytes32 }

State update transaction

struct StateUpdateTransaction { proof: Proof snark_account: Account new_height: u32 new_root: Bytes32 }

fn process_transaction(tx: StateUpdateTransaction) { require(tx.snark_account.last_height < tx.new_height); let vkey = tx.snark_account.vkey_stf; let trusted_header = self.header_at_height(tx.new_height); let public_inputs = [trusted_header tx.new_height, tx.snark_account.last_height, tx.new_root, tx.snark_account.state_root,]; verify_proof(public_inputs, tx.proof, vkey) // Update the account tx.snark_account.state_root = tx.new_root tx.snark_account.height = tx.new_height }

In the proof, the statement that is being verified is the following:

const namespace = "my_namespace" let namespace_blobs = [] let deposit_transactions = [] for height in last_height...new_height { let header = verify_header(trusted_header, header, height); let blobs = get_blobs(header, namespace); namespace_blobs.push(blobs);

let deposit_tx = get_deposit_tx(header, snark_acct_address);
deposit_transactions.push(deposit_tx);

}

verify_stf(state_root, new_root, namespace_blobs, deposit_transactions)

Note that in the above computation, the proof must

process ALL blocks between heights last_height

and new_height

to ensure that it is including ALL blobs for the relevant namespace when verifying the STF. This is done to ensure that the prover is not excluding any rollup blobs submitted to Celestia. Note that we must include all deposit transactions on Celestia to the SNARK account as this is an input for the STF for the rollup.

Cost Analysis (Circuit):

Note that even if a block has no blobs corresponding to a namespace, it requires 1 opening proofs for the NMT for proof of non-inclusion, which is non-zero cost across all the intermediate blocks.

In a SNARK, the cost for the above computation is:

(new_height - last_height) * (parent_header_opening + NMT_opening_proof) + STF_verification + B * NMT_opening_proof

if there are B

shares between (last_height, new_height)

for the rollup’s namespace.

The parent_header_opening

proof from our Tendermint implementation is ~6 SHA hashes and the NMT opening proof might be around 30 SHA hashes, for a total of H * 34

sha hashes for procesing H

celestia blocks, and a total cost of (34H + 30B) SHA_cost + STF_verification_cost

in terms of circuit complexity.

Withdrawal processing transaction

struct WithdrawalTransaction { proof: Proof snark_account: Account new_withdrawal_accumulator: Bytes32 withdrawal_account: Account amount: u32 withdrawal_id: Bytes32 }

fn process_transaction(tx: WithdrawalTransaction) { let vkey = tx.snark_account.vkey_withdrawal; let state_root = snark_account.state_root; let withdrawal_accumulator = snark_account.withdrawal_accumulator let public_inputs = [state_root, withdrawal_accumulator, tx.new_withdrawal_accumulator, tx.withdrawal_account, tx.amount, tx.withdrawal_id]; verify_proof(public_inputs, tx.proof, vkey) // Update the account tx.snark_account.withdrawal_accumulator = tx.new_withdrawal_accumulator

// send the amount to the withdrawal account
tx.withdrawal_account.balance += tx.amount
tx.snark_account.balance -= tx.amount

}

In the proof, the statement that is being verified is the following:

let withdrawal = (withdrawal_id, withdrawal_account, amount) verify_withdrawal_exists(state_root, withdrawal) verify_unspent(withdrawal_id, withdrawal_accumulator) verify_accumulator_update(withdrawal_id, withdrawal_accumulator, new_withdrawal_accumulator)

When the verification is complete, the SNARK account sends the withdrawal amount to the recipient and updates its accumulator.

Cost Analysis (Circuit)

Verifying existing of a withdrawal against the state root of the rollup is 1 merkle proof of the rollup state (can potentially be quite cheap if the rollup is using SNARK-friendly hash function for its state committment).

Verifying that the withdrawal is not in the current accumulator and updating the accumulator properly with the withdrawal id can be similarly cheap if the accumulator is constructed with a snark-friendly hash function.

Tradeoffs:

The above has the following drawbacks, with potential fixes or thoughts suggested in subbullets.

- The biggest tradeoff with this design is that because the user must know the current withdrawal accumulator to create a withdrawal transaction, without external coordination with the block builder, there is only 1 withdrawal per block.
- Requires 2 vkeys
- It would be easy to get rid of this if we combined the udpate_state_root

and process_withdrawal

circuits into 1, where a withdrawal transaction would also update the state root (if needed).

- It would be easy to get rid of this if we combined the udpate_state_root

and process_withdrawal

circuits into 1, where a withdrawal transaction would also update the state root (if needed).

- Keeps around a withdrawal accumulator and state root
- Seems hard to get rid of a withdrawal accumulator, since we need some way to keep track of the processed withdrawals, unless we want process all of the withdrawals sequentially
- Seems possible to get rid of state root with recursive snarks

- Seems hard to get rid of a withdrawal accumulator, since we need some way to keep track of the processed withdrawals, unless we want process all of the withdrawals sequentially
- Seems possible to get rid of state root with recursive snarks

- Keeps around last processed height from Celestia

- Seems possible to get rid of last processed height with recursive snarks
- Seems possible to get rid of last processed height with recursive snarks
- When processing state root updating transaction, it requires Celestia to provide the header root of a block height arbitrarily far in the past.
- Seems hard to get rid of this requirement because the proof has to be generated with respect to a particular Celestia block hash that may not be the tip of the chain
- Seems hard to get rid of this requirement because the proof has to be generated with respect to a particular Celestia block hash that may not be the tip of the chain
- The “update state root” circuit might be very expensive to prove because it requires a lot of SHAs for NMT opening proofs to get all of the share/blobs.

Option 2: Processing withdrawals sequentially

In this option, we only have a STF snark that also has as public input allpending withdrawals that occured between the last state root and the current state root, and as part of that transaction process all of the withdrawals and send them to the corresponding recipient accounts. Most of the other details are the same around the computation that the STF snark verifies, but the SNARK account would have less state:

Snark Account State

struct SnarkAccount { balance: u32 vkey_stf: VerifyingKey last_height: u32 state_root: bytes32 }

struct StateUpdateTransaction { proof: Proof snark_account: Account new_height: u32 new_root: Bytes32 all_withdrawals: Withdrawal[] }

fn process_transaction(tx: StateUpdateTransaction) { require(tx.snark_account.last_height < tx.new_height); let vkey = tx.snark_account.vkey_stf; let trusted_header = self.header_at_height(tx.new_height); let public_inputs = [trusted_header tx.new_height, tx.snark_account.last_height, tx.new_root, tx.snark_account.state_root, tx.all_withdrawals]; verify_proof(public_inputs, tx.proof, vkey) // Update the account tx.snark_account.state_root = tx.new_root tx.snark_account.height = tx.new_height }

- The verify_proof

would verify that all withdrawals included in the public inputs are the only

withdrawals between the previous rollup state root and the current rollup state root (can be done by checking the nonce of a withdrawal contract and ensure that it is always incremented by 1 or something similar).

- This would remove the need for the “withdrawal accumulator” field because it’s guaranteed all withdrawals are processed in order (and exactly once) and also remove the need for the withdrawal verification key. It’s also possible to remove the state_root

field with recursive snarks as described above.

- One con of this approach is there is a “free-rider” problem where the person submitting a proof for their particular withdrawal also has to pay the gas costs associated with all the withdrawals prior to them that have not been processed. But it is perhaps more elegant.

Option 3: Blob submission requires proof to update state root

One problem with the above design is that someone can spam a namespace with a bunch of garbage data that does not have valid transactions, but it might be very expensive to process this garbage data within a SNARK (for example if the data is very large). One method of preventing this type of attack is to require a proof alongside blob submission to a particular namespace that updates the STF alongside blob submission.

The SNARK account state would still be similar to the design above, but to post a transaction to a namespace with a snark account enabled would require that there is a valid proof to update the STF of the SNARK account with the new state root with the corresponding blob.

In this design, the snark account state could also have last_height

removed as we are guaranteed all submitted blobs are processed.

Snark Account State

struct SnarkAccount { balance: u32 vkey_stf: VerifyingKey vkey_withdrawal: VerifyingKey state_root: bytes32 withdrawal_accumulator: bytes32 }

The withdrawal process against the state root would look very similar to Option 1.

Tradeoffs

- The main drawback of this design (that likely makes it untenable) is that then the rollup's finality is bottlenecked by proof generation time because posting DA is coupled with a valid proof of the STF. The main advantage of Celestia right now is with the 12 second fast-finality rollups can post to the DA layer and have finality within 12 seconds, and proof generation can be slower but doesn't impact time to finality. This design would break that.

Option 4: The base layer provides replay protection for recent-history, user proves long-term

The high level idea is that user withdraws have a TTL of at most TTL_LENGTH

into the future. The base layer stores all succesful withdrawals for a period WITHDRAW_STORAGE_PERIOD > TTL_LENGTH

.

Every withdraw has an ID containing:

- a nonce
- A timestamp for the rollup's block time when the withdrawal was created

Snark Account State

struct SnarkAccount { balance: u32 vkey_stf: VerifyingKey vkey_withdrawal: VerifyingKey latest_state_root: bytes32 latest_withdrawal_accumulator: bytes32 // Timestamp -> (state root, with draw accumulator) storage_period_headers: Map storage_period_withdraws: Map }

When a withdraw request gets to the base layer, it consists of:

- ZKP Public input:
- Withdrawal ID (nonce, timestamp)
- Header to verify fields
- Timestamp, State root, withdraw accumulator
- Timestamp, State root, withdraw accumulator
- Withdrawal ID (nonce, timestamp)
- Header to verify fields
- Timestamp, State root, withdraw accumulator
- Timestamp, State root, withdraw accumulator

Additional base layer logic:

- Check that current block time is less than withdrawal.timestamp + TTL_LENGTH
- Check that withdrawal.nonce is not in snarkaccount.storage_period_withdraws.
- Set storage_period_withdraws[withdrawal_id.Nonce] = block time

Maintain state for every withdrawal ID in the last week. This is done with two state entries, one in the account one global across celestia (TBD how Celestia wants to deal with this, e.g. future withdraws have to prune 2 expired txs?):

- List of withdrawal ID's to prune at different heights.

Possible tx flows

Withdraw attempt on rollup occurs → Withdraw succeeds on Celestia

Withdraw attempt on rollup occurs → Fails to get onto Celestia in TTL window. Show this on rollup and get funds returned.

Header updates

Todo import logic thats described for STF verification, to allow adding more STF's that aren't just the latest STF.

State pruning options

- Every withdraw has to delete at least 2 ‘expired txs’
- Some global celestia logic to delete expired state

Other questions:

1. What ZK scheme should we use?
2. Is adding extra state beneficial to rollups? If there’s a special tx type to update the state of the rollups onchain, then they won’t have to recursively prove previous proofs
3. Anyone can post to a specific namespace - how do we avoid spam and filter for specific transactions? - solvable with intra-namespace blob prioritization

, where the rollup only process eg the 100 highest priority blobs (e.g. according to their gas paid). Based vs Sequenced rollups.

1. NMTs in Celestia use SHA256 - which are quite inefficient with ZK (although this can be optimized)
2. Comparison to ZK IBC, Bitcoin Opcodes, Mina rollups? Benefits of ZK IBC
3. How does gas work for these sorts of transactions.
4. Do we need a deposit accumulator? One Celestia tx could include multiple deposit txs to various rollup accounts. Events are emitted on Celestia and committed to block headers - similar to Ethereum
5. Proof size?