Always up-to-date draft is located here.

This construction may not be final and I'll continue to work on it for better UX and guarantees. Everyone is welcome to find the weak spots and errors in this construction so at the end of the day it can become correct and final!

Special thanks to josojo for an inspiration!

zkSNARKs governed Plasma by Matter Inc, informal spec

Draft 0.3, with Q&A and clarifications

This Plasma is not an UTXO one, but one with the balances, like the Ethereum itself!

Rationale

I'll skip the huge explainer about zkSNARKs here and give mainly two reasons why zkSNARKs may be a useful for Plasma constructions:

- prove the validity of state transition: State(block N) -> State(block N+1)
- hide most of the witness (like Merkle proofs) as private inputs

These options combined allow to reduce the proof to basically two G1

and one G2

elements (Groth16 proof system) plus O(1)

Fq

field elements.

The biggest challenge

Although one (here I mean the Plasma operator) can prove the validity of state transition, so the block is correctly formed, applied, etc, any Plasma with the global state (for example, with some form of tree for balances storage) is plagued by enormously difficult data availability problem. This problem for a trivial exits (using the Merkle proof of the balance for some block) can be illustrated as the following:

- There is a correct state at block N
- Mallory

starts an exit from block N

claiming a right to exit some balance

from the particular account

- An operator publishes a provably valid(!) block N+1

and withholds it

- In this block there is a transaction from Mallory

to her accomplice that spends Mallory's

balance

- Such transaction is a pure double-spend that can not be challenged due to data for challenge not being available
- Such chain of transactions can continue

Proposed construction

To even try to solve such a problem one can try to make a smart-contract that governs Plasma on the main chain much more complicated and strict for an operator. Unfortunately any challenge-response game with time limits is a potential attack on the operator in case of either miners' censorship or cryptokitties.

In parallel I'll calculate some rough estimates for number of constraints per transaction

in R1CS of state transition proving zkSNARKs. These estimates are based on:

- use of an embedded Edwards curve for signature verifications (roughly 4.2

constraints per bit of the field, multiplied by two for two scalar multiplications per signature check ~ 2000

constraints)

- use of the most constraint-wise effective hash functions:
- MiMC Longshift 2n/n

, ~ 800

constraints per hash

- Pedderrsen hashes based on the embedded Edwards, ~ 3000

constraints per hashing 2n -> n

- MiMC Longshift 2n/n

, ~ 800

constraints per hash

- Pedderrsen hashes based on the embedded Edwards, ~ 3000

constraints per hashing 2n -> n

For detailed information check the following sources[1], [2], [3].

State format

For purposes of this construction all the balances, public keys and additional information are stored in leafs of the sparse Merkle tree (SMT) of the depth D

. To form the final state

the root hash of SMT is additionally hashed with the block number that last updated this state. Each leaf stores the following parameters, that are later concatenated for hashing:

- public key
- public key of the onwer of this leaf
- balance
- nonce
- for replay protection
- last sent at height
- the chain height when there was a last transfer from this account

Transaction format

The transaction is an intent from user to send part of his balance to another participant of the system. Account numbers are limited to the range from 0

to 2**D - 1

, where D

is a depth of a sparse Merkle tree mentioned above, and numerate leafs in the SMT.

The transaction format is the following:

- from
- enumerates an account of the sender

- to
- enumerates an account of the recipient
- nonce
- should match the nonce of the from

account in SMT if this transaction is to be included in a block

- last sent at height
- indicates a corresponding field of the state. Logic is described below
- good up to height
- indicates that this transaction should only be included in a block of this height or less
- max fee
- the maximum fee amount that the sender wishes to pay for a transaction
- allow multiple in block
- a permission from user to include multiple transactions into block

max fee

field is optional in constructions that don’t use per-transaction fee.

Combination of separate nonce

and good up to height

is required as the Plasma is potentially censorable and user should have some guarantees that the Plasma operator will not have a power of second opinion and include an outdated transaction in a block.

Multiple transactions per block

User can grant a permission to an operator to include more than one transaction in block. While the largest portion of state transition proving zkSNARK is described in the next section, here is a rationale why last sent at height

is required as a part of the transaction and how multiple transaction per block work:

- By providing an explicit last sent at height

user gives a consent that he has observed a full block

where his last transaction was included.

- Of course there exist an options about partial or full block withholding where only an operator knows it and can continue to sent new transactions, but it will implicitly prevent normal users from doing so!
- If allow multiple in block

is NOT set an operator should check the following: * nonce

matches the data in SMT

- last sent at height

matches the data in SMT

- nonce

matches the data in SMT

- last sent at height

matches the data in SMT

- IF allow multiple in block

is set an operator should check the following: * nonce matches the data in SMT for every transaction in sequence in this block

- last sent at height

either matches the data in SMT OR is equal to the new state height

- nonce

matches the data in SMT for every transaction in sequence in this block

- last sent at height

either matches the data in SMT OR is equal to the new state height

good up to height

field logic is described in the next session and is valid for both set and unset flag allow multiple in block

.

Transactions block

Transaction block

is based on a zkSNARK that proves proper update of the sparse Merkle tree. It has the following public inputs

:

- Old state

.

- New state

.

- Merkle tree root of the transactions tree of the applied block.
- Total collected fee

Internally the following is proved per application of one transaction

:

- Block ordering is checked (old state height + 1 = new state height

)

- Using the Merkle proof internal states of the two leafs updated by the transaction are proved for the old state
- Content of the transaction

in block is a private input

- Ownership is properly checked, balances

are properly updated, nonce

is increased, last sent at height

is updated for sender

- Rules about last sent at height

for transaction with permission for multiple inclusion per block are honored

- good up to height

field of the transaction should be greater than or equal to the new state height

- new Merkle tree root is recalculated
- total fee

is updated

Procesure is repeated for the set of transactions, followed by calculation of the block Merkle tree root and the new state from the balances

Merkle tree root and the applied block number.

In terms of number of constraints to R1CS price of such update is clearly dominated by the term $4 \mathit{ConstraintsPerHash} D$

, where D

is a balances tree depth. For state tree of depth 32

it's roughly 1000 * 128 ~ 150000

constraints per one transaction application. One can limit a R1CS size to 1 billion

(DIZK FTW) that implies ~8000

transactions per block. Of course there are technological roadblocks - balances tree can not be updated concurrently, so transaction speed may be quite low.

The block is considered applied after an operator publishes a full content of transactions

from this block for public inspection, so everyone can update their states and compare with one published by an operator as a part of the proof.

Deposits block

Deposit block

updates the balances tree by inclusion of the new leaf instead of empty one with the proper associated public key, balance, nonce = 0

and properly set last sent at height

and last updated at height

. Size of such block is expected to be small, for example it should only serve up to 128 deposits. In such zkSNARK public inputs are the following:

- Old state
- New state
- All deposit balances and associated public keys

Although the word "all" here can be scary, all this information is already listed in the smart-contract upon deposit requests from users. One should also make a similar kind of topup block

to topup existing accounts, although those are skipped entirely in this document.

One important aspect for limitation of number of deposits to the Plasma: an operator should keep some security deposit that should cover the cost publishing an exit blocks

in the amount required to exit every balance, so with growing number of deposits into the new leafs an operator MUST increase his security deposit.

Exits block

Exit block

is quite simillar and it updates the balances tree by zeroing the balances of the leafs with the proper request owner.

Upon the exit request

the user provider a zkSNARK proof of the full data availability for the last state

his transaction was included in the chain along with some bond. This is effectively a Merkle proof for some chain height

. As public inputs

such zkSNARK discloses the nonce

, last sent at height

and last updated at height

for the exiting account

. The last sent at height

field is checked agains timestamping and if more than 3 days

has passes the request is accepted. Those requests form a queue based on the last sent at height

- the smaller is better.

Priority capping

is allowed here, where an exit from the very old state

with very small last sent at height

will have it's priority capped to the chain height

of - 1 day

from now, and this “capped priority” will be used as a counter for the data availability rollback described in the corresponding section. If priority capping is not introduced than there is a potential for infinite state rollback if a malicious party starts valid

exits with smaller and smaller last sent at height

that will always get into the exit queue before every other exit request.

This exit request can be challenged over the 3 days

period by providing a proof of full data availability that discloses a state

for the same account

with higher nonce

and last sent at height

. This is a griefing vector, although the user should only be griefed at maximum once if he properly checks the data availability and does not sent transactions if the latest committed block is not published in full.

If the exit request

is not challenged it has to be included into some exit block

no later than 1 day

after the end of the challenge period. This requirement is enforced by the smart-contract that manages the exit queue.

Public inputs

for the exit block

zkSNARK are the following:

- Old state
- New state
- All accounts

and last sent at height

and from requests for exits from the users

- All balances

for the corresponding exit requests. These balances are provided by an operator based on the old state.

Internally the following is checked:

- With a Merkle proof the latest internal state of the account

is revealed baseon on old state

- last sent at height

is checked againts the request data

Such requirements provide the following guarantee:

- If user tries to exit not from the latest state and data is publically available - he is challenged by other users
- If user tries to exit not from the latest state and data is NOT publically available - he is challenged by the operator (griefed), but this actions reveals a newer state
- If user is not challenged or exits from the latest state, an operator is enforced to process his exit or the chain is halted

Extra requirements for submission of different block kinds

One can further improve the user experience by adding more limitations:

- Total number of pending deposits should be limited
- An operator MUST produce a next block as a deposit block

if pending deposits queue is capped

- An operator MUST produce a deposit block

once every X

hours if there are any deposit

requests pending, otherwise the contract at first does not allow to send any more block, and later after some extra delay prevents any blocks from being published (effective stop)

- If the user's deposit was not included in a deposit

block after some delay T

a revocation procedure is allowed

- Total number of pending exits should NOT be limited
- Same way an operator MUST either produce an exit block

once every X

hours, or if exits limit is above M

items, or the block submission is stopped. In case of more than M

items in the queue an operator must produce the number of exit blocks

to pop those M

items from the queue

- Any kind block should be published at least once every X

hours and in case of the huge pending exits queue the queue above has to be popped over the perios of Y

hours, otherwise a block submission is prohibited

- Any violations (enforced stops, effectively) are punishable by operator's deposit slashing in half

A stopped block submission is equal to either an operator going offline or a block withholding attack where the smart-contract tries to protect user rights limiting the operator in what kind of blocks he can send, and an operator failing to do so.

Actions in case of block withholding

If user sees a block data being unavailable he submits a request for an exit. At some point the block submission will be prohibited and it implies, that the first (unchallenged) item in the exit queue has the last sent at height

that corresponds to roughly - 4 days

from now. Such height is the deemed the latest available.

Prohibited block submission

There should exit a mechanism to still allow users to exit block submission is stopped by the smart-contract's rules. As described above one can determine the last fully available block by inspecting the exit queue. In case of the stopped submission (may be after some extra time delay) the following procedure starts:

- Any party can send a proof for a new exit blocks that start to pop requests from the exit queue over some time slot of 1 hour

- Those items are ranked by size - on the setup phase an operator MUST make zkSNARKs for exit blocks

of various sizes

- At the end of the time slot of if the exit block

of the largest size is published - such block is accepted and becomes a new tip

- The party who's proof is used to update the chain tip does NOT need to publish any extra data - everything important is already in the public inputs

of the proof and other participats can update their state accordingly and continue in the next round

- Procedure repeats until the queue is empty

- A block publisher gets some funds from the remaining operator's security deposit

- One can make a procedure more interactive with commit-reveal scheme to reduce number of proofs checked on-chain, but this will require a deposit from the party that commits to provide some proof for an exit block

Q&A

- What is the block is not available in full?
- In this case no one can prove that some leaf in the SMT because the new state

depends on the latest root hash of the SMT and can not be calculated without every transaction in block. The exception is if the block is padded by empty transactions, but it's trivial.

- In this case no one can prove that some leaf in the SMT because the new state

depends on the latest root hash of the SMT and can not be calculated without every transaction in block. The exception is if the block is padded by empty transactions, but it's trivial.

- Why last sent at height

should match at the exit block? * Imaging the following if last sent at height

is not checked: * User A

has a state with last sent at height = N

and sends a transaction A => B

- This transaction is included in block N+1

- User colludes with an operator

- User publishes an exit request with the last sent at height = N

- No one can challenge this request

- An operator proceeds with an exit block

AFTER block N+1

and this block is accepted as last sent at height = N+1 != N

, but this field is not checked!

- An operator publishes an exit from B
- This is a double spend!
- User A

has a state with last sent at height = N

and sends a transaction A => B

- This transaction is included in block N+1
- User colludes with an operator
- User publishes an exit request with the last sent at height = N
- No one can challenge this request
- An operator proceeds with an exit block

AFTER block N+1

and this block is accepted as last sent at height = N+1 != N

, but this field is not checked!

- An operator publishes an exit from B
- This is a double spend!
- If last sent at height

is checked an operator can not proceed with exit block

for user A

that has priority N

and that will become the first in a queue even if an operator published all possible exit blocks

, but still will not challenge an exit request with a newer state!. So, N

becomes a last point of validity and data availability.

- Imaging the following if last sent at height

is not checked: * User A

has a state with last sent at height = N

and sends a transaction A => B

- This transaction is included in block N+1
- User colludes with an operator
- User publishes an exit request with the last sent at height = N
- No one can challenge this request
- An operator proceeds with an exit block

AFTER block N+1

and this block is accepted as last sent at height = N+1 != N

, but this field is not checked!

- An operator publishes an exit from B
- This is a double spend!
- User A

has a state with last sent at height = N

and sends a transaction A => B

- This transaction is included in block N+1
- User colludes with an operator
- User publishes an exit request with the last sent at height = N
- No one can challenge this request
- An operator proceeds with an exit block

AFTER block N+1

and this block is accepted as last sent at height = N+1 != N

, but this field is not checked!

- An operator publishes an exit from B
- This is a double spend!
- If last sent at height

is checked an operator can not proceed with exit block

for user A

that has priority N

and that will become the first in a queue even if an operator published all possible exit blocks

, but still will not challenge an exit request with a newer state!. So, N

becomes a last point of validity and data availability.

- Are blocks published?
- Block headers (which are Merkle tree roots of the transactions tree) are published as one of the public inputs to the transaction block

zkSNARK proof.

- Those headers are NOT stored on-chain
- Users should download a block in full from the operator and check off-chain that the header from downloaded block matches the published one.

- Block headers (which are Merkle tree roots of the transactions tree) are published as one of the public inputs to the transaction block

zkSNARK proof.

- Those headers are NOT stored on-chain
- Users should download a block in full from the operator and check off-chain that the header from downloaded block matches the published one.

- Is this the only way to revert the chain tip?
- No, there are others. For example, one can remove blocks one by one, but in this case an operator can make a lot of unavailable blocks before the reversion process starts and revert will be long and gas expensive.
- In principle one can make some protocol for users to vote (using proofs for availability for some random account

in a state) what is the latest state

known for evertone.

- No, there are others. For example, one can remove blocks one by one, but in this case an operator can make a lot of unavailable blocks before the reversion process starts and revert will be long and gas expensive.
- In principle one can make some protocol for users to vote (using proofs for availability for some random account

in a state) what is the latest state

known for evertone.

Authors

- Alex Vlasov