Special thanks to the roll_up team for their hard work on building a first PoC for verifiable snapps (snark-dapps) and special thanks to Felix and Ben for their invaluable contribution to this idea.

Plasma snapp:

TL;DR

The following specification outlines a new plasma version which utilizes snarks to prove its integrity and validity. Via an interlinking between exit requests and deposits with the correctness proof of a block - the snark -, we are able to specify an implementation without any need for exit challenge games and confirmation signatures. Unfortunately, the concept of exit queues is still needed and users need to be online to receive payments.

Removing the exit games and confirmation signatures allow us to remove much of the complexity of plasma, which is currently hindering the implementation of more sophisticated protocols beyond simple token-transfers. This proposed version will facilitate to integrate more protocols into plasma by making the snarks itself handling these protocol advancements.

Introduction:

Over the last half a year, there has been made tremendous advancements regarding fully verifiable plasma chains. New signature mechanism and hashing mechanism were found, helping to reduce proving times significantly.

These advancements enable the following with reasonable timings:

- storing the complete state of a plasma chain encoded as a StateRootHash

on ethereum

- this StateRootHash can be updated by a central operator by providing a snark proving a valid state transition
- A valid state transition is proven within the snark by opening one or several leaves of the merkle tree describing the current state, checking the user's signatures, doing predefined operations, updating the leaf and finally recalculate the stateRootHash.

How this can be in done in detail, you can find over here: [https://github.com/barryWhiteHat/roll_up

](https://github.com/barryWhiteHat/roll_up)

However, snarks do not solve the problems associated with data unavailability. Also, the snarks need to be aware of any incoming deposit request to the plasma chain and outgoing withdrawal requests. This post will describe a solution for these two remaining issues.

Proving valid state transactions

In this model, the state is described by leaves of the StateMerkleTree. Each leaf is a list of a

[public key, amount of token, block height of last transfer].

The plasma contract is aware of the current state, as we store it as the variable StateRootHash

in the contract. it also stores the verification keys of the snarks of 3 different programs: P_transfer, P_deposit, and P_exits for 3 different kinds of state changes. These verification keys allow the plasma contract verifying that the state changes for a new block are actually valid ones.

Let's call the program checking the correctness of a state transition based on transfer P_transfer:

P_transfer(StateRootHash_i, [witness transactions data]) = (StateRootHash_(i+1))

A transaction is a value transfer from one leaf to another leaf. P_transfers checks the following:

- Leaf of sending account exists
- Leaf of sending account has the needed balance for transfer
- The transfer transaction is signed by the private key associated with the public key stored in the sending leaf.
- Subtracts balance from sending leaf, update block height of last transfer, updates theStateRootHash
- Leaf of receiving account exists

- Updates balance and block height of last transfer of receiving account

The witness transaction data will be used by the snark proof, but they will never touch the rootchain. It is only needed to create once the witness of the snark.

This is pretty straightforward. But how do we make the plasma chain aware of deposits and withdrawal? We think the following trick will do the job:

The information of several deposits, which are sent to the plasma contract, can be hashed together to a depositHash

by the plasma contract. By requiring the snark proof to take the depositHash

as a public variable, we can enforce the snark proof to process all deposits.

The program P_deposit checking the correctness of deposits would look like this

P_deposit(StateRootHash_i, DepositHash_i, [witness deposit data]) = (StateRootHash_(i+1))

A deposit is a value transfer into an empty leaf. Whether a leaf is empty, we will store on the ethereum mainchain in a mapping: leafOccupation. P_deposit does the following:

- Insert public key and deposit amount into the leaf
- Updates the StateRootHash

The same can be done for exits: All exit requests can be collected by the plasma contract. The operator submits for each exit request the currently withdrawable balance from the plasma chain. Then these information are hashed together in an exitRequestHash

and by making it a public input the snark, we can enforce the snark to process this exit request.

P_exit(StateRootHash_i, ExitRequestHash_i, [witness exit data]) = (StateRootHash_(i+1))

A withdrawal tries to exit all balance out of a leaf. The snark checks the following:

- Leaf has not sent or received a transaction within the last 7 days. (If this would be the case, the operator would have to set the predefined withdraw balance to 0, and the snark would be exited.)
- Leaf currently stores exactly the predefined balance
- Deletes the complete leaf, updates StateRootHash

Unfortunately, exits with a fraction of the balance are not supported by this protocol.

Note that the operator needs to set the withdrawal balance, as only he knows for sure the current balance. Still, the snark enforces the operator to set the correct balance, as otherwise he will not be able to find a proof. If someone makes an exit request, which is not valid, the operator will set the balance in the exit to 0. Exit request against non-occupied leaves are prevented by the plasma smart contract.

These three different programs allow the operator to append the plasma chain with 3 different block types (deposits, transfers and exits) by sending over the respective snark prover key. The plasma smart contract would enforce that registered pending exits would be processes at first, forcing the operator to submit exit blocks, before deposit or blocking transfer blocks. Likewise, if there are deposits pending since several blocks, then the plasma contract would force the plasma operator to include deposit blocks, before any transfer blocks are accepted. Only if there are no pending deposits or withdrawals, then the plasma chain allows appending transfers blocks.

Roll back of the tip of unavailable chains

This above construction interlinks very well deposits and exits with the snark proofs. Unfortunately, it can not prevent the data unavailability case. It could always happen that the operator would publish a StateRootHash and nobody knows the content of this new state. Using priority queues for exits and an unwinding mechanism, we can solve the problem:

If the chain operator stops publishing new valid blocks for 3 days, then the plasma root chain contract would allow swapping the operators. Then anyone else can extend the plasma snapp chain provided that the new operator hands in valid snark for his new blocks.

If no one can build on the tip of the plasma chain, then the last block of the plasma chain will be removed and we wait for people building on the second-highest block. We will continue this removal process of the plasma chain tip, until it is extended again by another operator. If clients are storing all the data of the plasma chain, then they could always become the operator themselves in such a situation. Thus, if they do not agree with the reversal of a block of the chain tip, they need to become the operator. This is a fair mechanism, as everyone can stop the roll back of the history.

Payment receivers should only acknowledge their payment as accepted, once they have the complete new state of the plasma chain. Then they could theoretically always prevent the roll back of their received transaction by becoming the plasma operator themselves.

Becoming a plasma operator is a heavy task, but on the other side, we could also incentivise this heavily by slashing the original operator and rewarding the new operator with these slashed funds: During the plasma chain creation, the operator has to make a deposit of x ether. If the plasma chain was stopped and the operator swapped, then the new operator would receive a fraction of this ether per submitted valid block. This ensures that the plasma chain is continued for quite some time after the original operator was switched and everyone has the chance to leave the chain.

But there is one hook, unwinding transactions might be fine, but unwinding withdrawals is not possible. However, there is a workaround: We require that

- exit requests are only included in the ExitRequestHasH

of the plasma block at least 40320 blocks (7 days) later after their account was sending the last transaction.

- users initiate their exit request at the latest 40320 blocks (7 days) after they see the unavailability of the data

If a users wants to withdraw, he sends a request to the plasma root-contract with the blocknr: blocknr of last touch. The plasma contract then requires the snark to process this exit exactly at min(blocknr + 40320, current block + 1). Especially the plasma smart contract will make sure that the transfer block: blocknr + 40320 is not accepted before the exit is processed.

Using this delayed mechanism, users funds are safe:

Imagine the block n is the first unavailable block.

1. If the operator keeps on building blocks, but stops building new blocks before the block n+40320, then all exits processed must be “old” transaction, which are anyways were not touched since the data unavailability. Even when blocks are unrolled until n, we require the new blocks to include the same exits, but without paying for the exits again. Thus no funds are lost.

2. If the operator keeps on building blocks and goes beyond the block n+40320, then every user should have already registered their exit and it should be withdrawn.

Note: In order to make this mechanism work, everyone needs to know when their transaction might have been included into the plasma chain. Especially, the scenario needs to be prevented where the operator produces unavailable blocks and includes in these blocks old transactions of somebody and thereby prevents this person to withdraw. Hence, every transaction submitted by a customer should be valid only for a specific block height.

Note2: A malicious operator could top up the balance of other users to touch their leaves in the unavailable block, and thereby give regular users a bad priority in the exit queue. This attack vector can be migrated by requiring the operator to hand over to the snark as witness a message signed from the receiving party: " I have seen block x and I am okay with receiving funds in block x+1".

If someone could come up with a better solution, which does not require the receiver to submit this message, this would be great

Note3: This specification has the nice property that we can use any hash function for the state merkle tree within the snarks, as these hash functions do not need to be executed in the evm at all. This is a huge benefit. Only the deposits and exits need to be done with kecca, as here the evm will need to execute them as well.