TL;DR

In the last plasma call, one of the topics was a snark/stark-plasma chains, which can always be proven to have only valid blocks. Vitalik elaborates that one problem, which can still not be solved with snark/stark secured chains during withdrawals, is:

“the problem is basically that the system has no idea that the coins exiting at the different time might actually be the same ones”.

I made exactly the same observation, but found an interesting workaround: it relies on a tricky interlinking between deposit/withdraw requests and the plasma chain.

In the following, I would like to give a specification for a whole plasma chain, but first, let me give you a glimpse of the trick:

Setup description:

Let’s assume that the plasma chain at height N has a state S_n, which is described by variables s1,s2,…,sk (each s would, for example, represent a valid spendable utxo). We also have a program P that can check that a transaction t_i applied on S_n is valid and will generate the new state S_n+1. In order to have it more compact, P could check the validity of t_i in the following way:

P(i=(merkle_root(S_n)), w=(s1,..,sk, t_i)) =(merkle_root(S_n+1))

where i is the input of the snark, and w=(s1,…,sk,t_i) is the witness of the snark. The witness would entail all the variables describing the state S_N and the new transaction t_i.

P would essentially do the following steps:

1. calculate the merkle_root of (s1,…,sk) and check that it is indeed the input i.
2. would apply the transaction t_i on the state: (s1,…,sk) and get a new state (r1,…, rk)
3. would hash the state (r1,…, rk) and would provide its Merkle root as the output (merkle_root(S_n+1))

For simplicity, I wrote it down with one transaction, but a proof can be made for a transaction set as well.

This is the usual setup for snark plasma chains, but now we make it more complicated:

Idea: Interlinking plasma chain exits with root chain.

We assume that the plasma contract on the root chain will compactify all exits requests for a given ethereum block by calculating: newExitHash = sha256(newExitHash, exitUTXO)

. The plasma chain operator would be required to reference NewExitHash in one of his next snark-proofs. If a list of exits [exitUTXO_1, … exitUTXO_L] has generate the exit hash newExitHash, then the chain operator would need to provide a proof for it:

P(i=(merkle_root(S_n), newExitHash, exitIsValid), w=(s1,..,sk, t_i, [exitUTXO_1, ... exitUTXO_L])) =(merkle_root(S_n+1))

where exitIsValid is an array of zeros and ones indicating the validity(1) or non-validity(0) of the exit request. We require the program P to check that:

1. calculate the merkle_root of (s1,...,sk)

and check that it is indeed the input i.

1.1 calculate the iterative hash sha256(newExitHash, exitUTXO_i)

for i=1,…,L and check that the hash equals newExitHash from the input i.

1. For each exit in [exitUTXO_1, ... exitUTXO_L]
2. check that the exit request exitUTXO_I has the same validity if applied to the current state (s1, ...,sk)

, as indicated in exitIsValid[i]

- if the exit is valid, it will be applied to the state (s1, ...,sk)

and calculate the new state (s1, ...,sk)

by removing the exited utxo.

1. check that the exit request exitUTXO_I has the same validity if applied to the current state (s1, ...,sk)

, as indicated in exitIsValid[i]

1. if the exit is valid, it will be applied to the state (s1, ...,sk)

and calculate the new state (s1, ...,sk)

by removing the exited utxo.

1. would apply the transaction t_i on the latest state after the exit request processing:(s1,...,sk)

and get a new state (r1,...., rk)

1. would hash the state (r1,...., rk)

and would provide its Merkle root as the output (merkle_root(S_n+1))

We would only accept new plasma blocks of the chain operator, if he includes and proves his inclusion of exits in the latest plasma block. In order to avoid problems with reorgs in ethereum, he would be required to include the newExitHashes of ethereum blocks only after a certain delay.

The beauty of this construction is that the operator can still produce unavailable blocks, but he would be forced to processes all valid exits from the plasma chain. If he does not include them, then we have a proof that he did not do it and the root-contract will prevent the chain operator to submit new blocks. Then the plasma chain stops. In this case, we can still do exits from the last checkpoint and slash/punish the chain operator heavily.

Main observation:

Before working with starks, data-unavailability was always a subjective matter. Now, by linking plasma chain and root-chain exits, we get an objective criterion for the data unavailability. If there is a real data unavailability, the chain will halt. If there is only a partial data-unavailability for some parties, these parties can leave without any risk, as the operator is forced to validate their exits.

Outline of a specification for a plasma chain:

This specification uses the above-mentioned trick to build a very secure and light client friendly token-transfer plasma chain, which allows huge scalability. The scalability is only limited by snark/stark proof generation constraints.

We describe a plasma chain at height N with a state S_n

, which is described by variables [s]=s1,s2,…,sk. Each element in [s] is an address and a balance associated to this address. After each creation of a new plasma block, the current state of the plasma chain will be compactified as a hash and submitted to the root-chain. Along with the hash, we also submit a proof that the state transition from S_n to S_n+1, by applying the transactions of the plasma block on S_n, is valid and S_n+1 will be represented by the new hash committed to the root chain.

Deposits into the plasma chain will be compactified by the root-contract in a hash newDepositsHash

. Likewise, normal withdrawal requests from the plasma chain will be compactified by the root-contract in a hash newExitHash

.

The proof for each new block will the look like:

P(i=(merkle_root(S_n), newExitHash, [exitIsValid], newDepositsHash), w=([s], [t], [exits],[deposits])) =(merkle_root(S_n+1)),

where [s] is an array of the variables describing the current state, [t] is an array of transactions, [exits] is an array of exit data from the root-contract and [deposits] is the array describing the deposits data.

P would do the following checks:

1. merkle_root([s]) == merkle_root(S_n)
2. newExitHash == reiterated_hash([exits])
3. newDepositHash == reiterated_hash([deposits])
4. validates the list [exitIsValid] and updates the state [s] by processing each exit
5. validates all transactions from [t] and updates the state after each transaction
6. calculate (merkle_root(S_n+1))

This gives us a great plasma chain, in case we would never run into data-unavailability. In order to handle data-unavailability, we need some more inputs:

The chain operator needs to make a deposit of X Ether, before chain creation. He will only get them back, once he has processed all exits, such that the plasma root-contract does no longer hold any clients funds. In order to make this a viable solution, the plasma operator can also initiate valid exits.

These X Ether are a huge incentive for the plasma chain operator to keep the system alive. If he gets himself into data-unavailability, the plasma root-contract will notice it and will slash his X Ether.

We would also need checkpoints, as in case of a data-unavailability, all users would be required to withdraw from the plasma chain at the same height. So each day the chain operator marks one block as a checkpoint-block. Clients of the plasma chain would be required to be online once a day. Each day the client would ask the chain operator for a Merkle proof of his balance at the last checkpointed-block. If he gets this Merkle proof, all is good. If he does not get it, he will need to exit the chain by registering his account for an exit on the ethereum root-chain. Now there are two possibilities:

1. the operator exits his funds. Then the client is safe.
2. the operator does not exit his funds, but then the plasma-chain will stop. If the chain is halted, all users need to withdraw from the second last checkpoint, which was from the previous day. Every user would have a valid Merkle proof to withdraw their balance from the check-pointed block, since otherwise, they would have stopped the chain beforehand. Hence all users can exit the chain at the checkpointed block by making an exit request on the root-chain and by providing their Merkle-proof.

This means that the transactions of the last two days might be reverted.

In order to make this scenario unprofitable for the chain operator, we would require every stark proof to prove additionally that not more than X Ether has been spent, since the last two checkpoints. If this proof is in place, the chain operator would lose X Ether in case of chain stop and might get some gains smaller than X Ether by the reverting of transactions. Thus, the attack would be unprofitable for him in ANY case, and he would never intentionally introduce a data-unavailability.

This means, we can set up the plasma chains, such that data-unavailability will be heavily disincentivized. For me, this is pretty exciting stuff. I mean on top of this incentive to keep the plasma chain alive, we get great scalability and very fast exits “for free”.

Please be aware that this specification cannot yet be implemented, as snarks/starks proof calculations are still too inefficient. But I am sure that zero-knowledge protocols will evolve. Especially, recursive proofs attempts are looking quite promising.

*edits: I corrected some grammar