Authors BarryWhitehat, Alex Gluchowski, HarryR, Yondon Fu, Philippe Castonguay

Overview

A snark based side chain is introduced. It requires constant gas per state transition independent of the number of transactions included in each transition. This limits scalability at the size of snark that can be economically proven, rather than the gasBlockLimit/gasPerTx

as proposed previous.

Given a malicious operator (the worst case), the system degrades to an on-chain token. A malicious operator cannot steal funds and cannot deprive people of their funds for any meaningful amount of time.

If data become unavailable the operator can be replaced, we can roll back to a previously valid state (exiting users upon request) and continue from that state with a new operator.

System roles

We have two roles

1. the users, create transactions to update the state
2. the operator, uses snarks to aggregate these transactions into single on-chain updates.

They use a smart contract to interact. We have a list of items in a merkle tree that relates a public key (the owner

) to a non-fungible token.

Tokens can be withdrawn, but only once.

In snark transactions

The users create transactions to update the ownership of the tokens which are sent to the operator off-chain. The operator creates a proof that a

1. previous state
2. set of transactions

produce the newState

. We then verify the proof inside the EVM and update the merkle root if and only if the proof is valid.

Priority queue

The users are also able to request a withdrawal at the smart contract level. If the operator fails to serve this queue inside a given time limit, we assume data is unavailable. We slash the operator and begin looking for a new opeartor.

It is impossible to withdraw the same leaf twice as on a withdrawl we store each leaf that has been exited and check this for all future exits.

Operator auction

If a previous operator has been slashed we begin the search for a new operator. We have an auction where users bid for the right to be the operator giving a deposit and the state from which they wish to begin.

After a certain time, the new operator will be selected based on the highest bid (above a certain minimum) with the most recent sidechain state.

Roll back

When the operator is changed, we allow users to exit. The only reason a user needs to do this is if they got their token in a state that will be rolled back.

We order these withdrawals by state and roll back the chain processing transactions in that state until we get to the state where the new operator will continue from.

Note that since it is impossible to withdraw the same leaf twice, user cannot exit the same leaf from an older state.

Discussion

The operator is forced to process requests in the priority queue, otherwise they are slashed. If they refuse to operate the snark side of the system they are still forced to allow priority queue exits. Therefore, the system will degrade to an on-chain token if the operator becomes malicious.

A new operator should only join from a state for which they possess all the data. If not, they could be slashed by a priority queue request they can't process.

A user should not accept a leaf as being transfered unless all the data of the chain is available so that they know in the worst case they can become the new operator if a roll back happens.

It is probably outside the regular users power to become an operator, but their coin will be safe as long as there is a single honest operator who wants to take over. Also bidding on a newer state will give them an advantage over all other bids.

This however allows the current operator to again become an operator because they will know the data of the most recent state and can bid on the latest state. However we can define a minimum stake that will be slashed again if they again refuse to serve the priority que. Therefor we can guarantee that someone will come forward to process the queue or else the chain will roll back to state 0 and users can exit as we roll back.

Unlike Plasma constructions that cannot guarantee the validity of all states, this design avoids competitive withdrawals since snarks disallow invalid state transitions. As a result, we can recover from scenarios with malicious operators without forcing all users to exit (however those that wish to exit can still do so).

Appendix

Appendix 1 Calculations of tps

Currently it takes ~500k constraints per signature. With optimization we think we can reduce this to 2k constraints.

Currently our hash function (sha256) costs 50k transactions per second. We can replace this with pedersen commitments which cost 1k constraints.

If we make our merkle tree 29 layers we can hold 536,870,912 leaves.

For each transaction we must

1. Confirm the signatures = 2k constraints
2. Confirm the old leaf is in the tree = 1k * 29 = 29k constraints
3. Add the new leaf and recalculate the root = 1k * 29 = 29k constraints

That equals 60k constraints per transaction.

wu et al report that they can prove a snark with 1 billion gates.

1000000000 / 60,000 = 16666 transactions per snark confirmation

Validating a snark takes 500k gas and we have 8 million gas per block. Which means we can include 16 such updates per block.

16666 * 16 = 266656 transactions per block

266656 / 15 = 17777 transactions per second.

We can probably reach bigger tps rates than this by building bigger clusters than they have.

Note: running hardware to reach this rate may be quite expensive, but at the same time much less than the current block reward.