To improve censorship resistance (specifically, make censorship distinguishable from latencies longer than 1 slot), it seems desirable to allow some form of delayed block inclusion, where a block published at slot N

can be included at some slot N+k

. However, this goes against the basic stateless EE design, where every block must provide a witness, and that witness must be rooted in the most recent state root. Here I propose a modified EE design that gets around this problem, allowing delayed-included blocks to be processed correctly.

Basic proposal

If a block with transaction data D_T

is published with a state root S_{old}

that is not equal to the most recent state root S

, then that block is a no-op, except a record of this fact is saved. The next block is a no-op unless it includes the same data D_T

, but with a witness rooted in S

rather than S_{old}

.

We know that it will always be possible to use the information in the old witness, plus data provided in blocks between S_{old}

and S

, to reconstruct such a witness.

Note that if the next block is empty or missing or otherwise fails to do this, then subsequent blocks will continue failing until a block that correctly re-processes D_T

is included. Note that the first subsequent block to succeed could itself

be a delayed-inclusion block; unless some other block that re-processes D_T

is provided, the state root will not change in the meantime, so such a delayed-inclusion block would be able to re-process D_T

with no issues.

Optimizations

In the case of state that is read, but not written (eg. contract code), we can try to make some optimizations that will allow us to take advantage of proofs provided in the old block to verify data in the new block, removing the need to repeat the entire proof. This can only be done for state that is not modified, because if the state is modified then the full proof will be required to compute the new state root.

We can make a special purpose modification for the most common case (small-k-block-duration “uncles”, where S_{old}

is a recent ancestor of S

) as follows. When we are finished processing a block, we save a receipt, consisting of the post-states of all accounts that have been read during that block. This then allows a replacement block to work as follows: we provide the receipt for the S_{old}

block, including addresses and post-states, along with a receipt for every block between S_{old}

and S

, providing only the addresses.

The efficiency analysis is as follows: t

reads in an N-item state tree take t * log(\frac{n}{t})

hashes. In a replacement block, we simply provide the entire post-state receipts from the previous k

blocks, which is t*k

hashes. Hence, as long as k < log(\frac{n}{t})

(perhaps in practice k < \frac{1}{2}log(\frac{n}{t})

if we assume as many reads as writes), this improves efficiency. The extra space can then be used to include new transactions.