The interactive verification protocol Truebit recently launched. This post will assume that something like Truebit exists as a black box giving delayed results to queries of the form “run some code and return the output”, and show how to build an EVM optimistic rollup on top of it.

Truebit accepts code in WebAssembly. There are many Ethereum clients written in many languages, and many of them have compilation paths to WASM (eg. go, Java, Rust). The first step would be to create a “stateless” version of a client. This is actually easy: replace the database lookups with lookups to a table that is provided as an input. The resulting “client” would implement a pure function: process_block(state_lookup_table, block) -> post_state_root

. We assume that this can be compiled to WASM code.

The second step is to build the actual chain module. There is one challenge here: a blockchain is not

stateless. When a fraud proving process is started against some block N in an optimistic rollup chain, there is an implicit assumption that the state committed to by the pre-state root of block N is available (ie. common knowledge). For this reason, if someone sees that an optimistic rollup chain contains invalid blocks, they should challenge the first

invalid block. But Truebit by itself is an interactive computation system for pure functions. We get around this by simply moving a couple of easy steps of the interactive verification process outside of the Truebit call.

Here is a basic protocol description:

- The contract stores a chain of block hashes and state roots: List[Tuple[block_hash, state_root]]
- A sequencer (we leave it to the implementer to decide who is a valid sequencer; there may be multiple sequencers) can add a new block; they call a function add_block(expected_pre_state: bytes32, block: bytes, post_state: bytes32)

which requires the expected_pre_state

to be the head state root and then adds ((block, post_state))

to the chain.

- To challenge a state root, some challenger calls challenge(index: int, lookup_table: bytes, block: bytes)

. This: * Checks that the hash of the block

matches the saved hash

- Starts a Truebit call to process_block
- Computes and saves the Merkle root of the lookup table.
- Checks that the hash of the block

matches the saved hash

- Starts a Truebit call to process_block
- Computes and saves the Merkle root of the lookup table.
- Once the challenge has begun, anyone can challenge the challenger by proving that their lookup table is wrong: they can provide a Merkle branch rooted in the pre-state of that block showing a value at some position in the tree, and a Merkle branch rooted in the lookup table root showing that same value. If the two branches conflict, the challenge is cancelled and the challenger’s deposit is slashed.
- Once the truebit call returns the post_state_root

(the mechanics of Truebit itself ensure that this can only happen after at least one waiting period), it’s assumed that if there existed a valid challenge to the lookup table someone would have made the challenge, and so the remaining logic proceeds under the assumption that the Truebit result was correct. * If the result is not the previously-saved post_state_root

, and the result is not ERROR: LOOKUP_TABLE_MISSING_NEEDED_VALUE

, the challenge is successful, the original submitter is slashed, and the next few submitters, instead of publishing blocks, will be tasked with publishing corrected state roots to replace the incorrect state root and all subsequent state roots. A variable index_of_next_state_root_to_replace

is used to keep track of this process.

- If the result is

one of those two values, the challenger is slashed.

- If the result is not the previously-saved post_state_root

, and the result is not ERROR: LOOKUP_TABLE_MISSING_NEEDED_VALUE

, the challenge is successful, the original submitter is slashed, and the next few submitters, instead of publishing blocks, will be tasked with publishing corrected state roots to replace the incorrect state root and all subsequent state roots. A variable index_of_next_state_root_to_replace

is used to keep track of this process.

- If the result is

one of those two values, the challenger is slashed.

Note: if you make an invalid block, or

you challenge a block with an untrusted pre-state, you are vulnerable to getting slashed. So it's important for sequencers to add blocks only on top of a fully valid chain, and for challengers to only challenge the first

invalid block (so its pre-state is still guaranteed to be what you think it is).

Reminder 2021.05.03: ethresear.ch is a special-purpose scientific forum, and is not a general discussion venue for (especially non-technical) issues about crypto projects, even if

those issues are important. Please stay on topic.