Plasmabits

This is a proposal for a Plasma sidechain implementation that can run EVM-based smart contracts. In Plasmabits, the operator (a centralized block issuer) must do its best to keep the liveness, availability and correctness of the sidechain, or lose a security deposit on the root chain. This is similar to a proof of authority sidechain, as it can achieve sub-second times between blocks, but it's completely validated on the root chain. We deal with the risk of block withholding using preimage challenges, and validation of consensus rules is enforced using a system of challenges, and a TrueBit-like verification for the EVM execution. All these challenges are incentivized with a bounty, coming from a security deposit.

The main motivator for this is to run software as similar as what we currently have, without needing special plasma light clients or nodes. The security deposit makes it easier to estimate the security of the sidechain in monetary terms. I wasn't sure about calling this plasma, as most designs out there are UTXO-based and this is state-based, but it keeps (at least I think so) the assurance that it's safe under block withholding.

It's my first post here, so please let me know of any etiquette mistakes

Challenges

When working correctly, the operator frequently commits to the sidechain blocks in the root chain. A set of validations keep the operator honest. There are a number of insurance contracts incentivizing the verification of the chain. These contracts combined would make for a complete set of consensus validation rules (an actual virtual client) on the root blockchain. Such rules include:

- Withholding challenges: The operator might have submitted blocks to the blockchain, but it withheld the contents. The operator must present a preimage or get slashed
- Parsing challenges: The operator submitted an invalid block structure.
- Transaction censorship: Submit a transaction on the root chain, requesting for it to be included in the sidechain within a certain timeframe
- Invalid block signature: The operator provided an invalid signature of the block.
- Invalid previous block hash, height, or previous state, among other block verifications.
- Any other consensus failure checks, like transaction receipts posting an invalid after state.
- Invalid transaction execution: Need an on-chain way to verify a transaction.

The last step is the most complex technically, but using a Truebit-like binary search, we would only need to verify one EVM state transition. A PASITO contract (Plasma Arbitration Stepping Instruction Test Operator) computes such transition.

Block commitments to the root chain & Withholding

With a certain frequency (to be parametrized), the operator submits to the blockchain the root hash of a Merkle tree with all the block headers since the last checkpoint, plus what range of block heights it is committing to. The validators can construct such tree and verify that they are following the same chain.

In case the validator created a fork, this commitment serves as a proof to show the existence of two blocks with the same height (because the twin block would be signed with the operator's private key). There are some data structures that could save some storage space on the long run, like a Merkle Mountain Range. This commitment can also be used to challenge downtime of the operator (at least one submission every X time). In case the operator submitted a Merkle tree that validators can't create, a challenge can be created to query for a particular leaf value and its path. Weaknesses with this are network congestion of the root chain.

In order to validate that the operator is not withholding blocks, any validator might request the operator to submit a preimage of the hash of the block. Blocks on the sidechain would be limited in terms of size to whatever can be sent as data of a transaction in the root chain, accounting for the cost of hashing and verifying the hash. This is not so much a scalability problem, as blocks in the sidechain could have 1 tx per block.

Sidechain Halting

For any successful challenge, the contract tracking the status of the sidechain would stop accepting any further updates. This starts a challenge period where there needs to be certainty on which is the first block that broke consensus (the first invalid/withheld block, or the previous block from a fork) through a series of new challenges. But even in the case that an on-chain challenge is happening, other users and validators could decide to carry on if they don't agree with the challenge. Validating nodes could continue operating, if they trust that the challenge will eventually be resolved. Some challenges could even be solved by the validators themselves, as that they might want to mitigate the risk of a spam attack or miner censorship attack (or even the operator attacking itself).

Entering and Exiting

This is externalized to the contracts that want to do operations crossing the boundaries of both chains. The root chain hash needs to be included on the side chain, in order to facilitate atomic swaps between the sidechain and the root chain. In this case, the ETH that enters the plasma chain could behave like an ERC20. Exiting a halted sidechain would also be left to the cross-chain contracts, as users can provide proof of the world state of the sidechain before it came to a halt.

Entering

1. Alice deposits ETH in contract A. Contract A provides an uuid for this deposit and maps it to the amount.
2. Upon enough confirmations of the root chain, a block in the sidechain will include the root state of the main chain block where the deposit happened.
3. Alice can create a proof of state of the mainchain's contract A, for her deposit's UUID.
4. The contract B, in the side chain, issues an equivalent amount of tokens to Alice upon her submission of the proof. Marks the uuid as claimed.

Exiting (not halted):

1. Alice sends her sidechain-ETH to the contract B. The deposit gets burned.
2. Once the challenge period for withholding and other checks has elapsed, Alice can show proof of the burn in the sidechain to the contract in the mainchain and retrieve her ETH.

Exiting while Block withholding:

Assuming Alice is running a fully validating sidechain node, she should have all the blocks up to the moment when she did the burn, so she can submit the blocks herself (they are signed by the operator). If the block with the burn is withheld, and the operator fails to show proof of it in the mainchain, then the sidechain will halt, and Alice can follow the steps described in the next subsection. Otherwise, if the block next to her burn is withheld, the mainchain contract would handle this as in the previous subsection, and Alice won't be able to double-spend because her balance at the moment of the halt will be zero. In any case, Alice should not consider transactions finalized until receiving the block with the transaction, and making sure that all blocks before are valid and available (she can respond to challenges herself).

Exiting (halted):

1. Alice shows proof of the world state in the sidechain at the block when the sidechain halted.
2. Contract A returns the equivalent amount of ETH, and marks that the address can no longer withdraw (which is fine since the sidechain is halted).

Also, there could be a "Never exiting" policy, where a new sidechain can pick up from where the previous operator left, as specified in the root chain contract (perhaps some oracle, or an auction for who creates the biggest security deposit).

It could have some constraints to accommodate for security margins and challenge periods.

PASITO: EVM²

This past October, a precompile was suggested to run theEVM inside an EVM. We intend to do the same, by having a stepper contract that can compute a EVM state transition.

Some work on this already started (seesolevm), but we want to focus on correctly encoding the whole EVM state in such a way that it can fit inside a transaction in the root chain, for the purposes of verifying it with an interactive Truebit game. We believe that a large security deposit, plus other economic interests that participants might have in the correct operation of the sidechain, would lead to less risks than general purpose Truebit (sorry about the hand-waviness here – this needs more careful analysis).

Some elements of the EVM state could be submitted under a hash tree structure to save transaction bytes, like unaccessed parts of the code, memory, and the stack if it grows too much. The final truebit transaction would have to provide witnesses for storage as needed.

Potentially, all the *COPY operations would be the most problematic opcodes – because creating a succinct proof for highly partitioned memory can be very challenging (would need to have a nice data structure to produce offset-friendly-merkle-trees to calculate quickly the overlay of one array on top of another). Overlaying multiple merkle proofs can do the trick, but this is still early work in progress (it could be a list where its elements have a range and the merkle root for each range; so retrieving a value from here is achieved by cycling through this list until the range matches, and finding the value in the merkle tree).

Something that is pretty interesting is that running only one step makes things like snapshots or rollbacks an easier problem than with a normal interpreter, so actually the CALL ops are not very difficult.

Economic incentives for validation

To prevent front-running, all challenges should follow a commit-reveal schema. But validators that are not miners could be still be exposed to front-running, thus reducing the incentive to validate the chain if one doesn't have a root chain mining operation (a miner could not validate the sidechain until it sees a challenge, and censor the transaction until it can see the mistake in the execution of the EVM or a similar error). This is one of the biggest blockers and where we would really appreciate feedback.

On the subject of allowing several challenges at the same time, the first committed-to challenge would win, but further research on what kinds of situations this can bring is required.

Withholding challenges would have to have a cost, but the dynamic of this is a little weak - either the users could grief the operator, or the operator grief the users by always withholding until challenged.

Ideas / Difficulties

- Economic incentives for the slashing/rewarding of the validators: would like to provide the security deposit in full to the successful challenger, but to prevent the operator from challenging itself, burning a percentage (say, half of the deposit) looks like a better idea.
- The EVM stepper has to be completely compatible with geth/parity execution
- Build a validating client that can generate proofs
- Complexity of the evm state transition contract
- Network attacks (spam, miners, high transaction fees, high cost of answering challenges)
- Considering ewasm instead of evm
- Complexity of the transaction cleanup (correct receipts included in the block and state updates)

I've been trying to get as much feedback as I could, and I feel this is a fairly viable solution in the short term. Would love to hear feedback on potential blockers. Our next steps involve focusing on clarifying some parameters, like time between commitments, complexity in the operation of validator code, complexity of proofing the current instruction to run, instantiating the EVM state, and more.

Thanks @jdkanani, @johba, @federicobond and others for reading earlier versions of this and gifting me your valuable input.

Some related posts: SMART Plasma proposal, Plasma checkpoint cost and block time, A cryptoeconomic accumulator for state-minimised contracts