Introduction

The Plasma MVP implementation expects users or third party to actively monitor and challenge the exit attempts. However, in reality many users are offline most of the time. Such an assumption could be a bit strong in practice.

In this post I would like to explore the idea of using an on-chain smart contract to probabilistically verify the off-chain Plasma state for a special yet widely applicable scenario, where the users can be divided into two groups, a “payment sender” group and a “payment recipient” group. Payments mostly flow from the sender group to the recipient group

. Such division can be found in many real world applications:

- Video games

. Here the recipient group contains just the game publisher, and the sender group consists of the gamers, who sends payments to game publisher for in-game virtual items. It is reasonable to assume the game publisher to be online all the time, while the gamers are offline when they are not playing. Typically, a gamer would deposit a certain amount of money (e.g. $50) into a game, and then buy different in-game items with multiple micro-transactions.

- Online marketplaces like Amazon

. Here the merchants form the recipient group, while the individual buyers constitute the sender group. The majority of the payments flow from the individual buyers to the merchants. The reversed payment flow – refund – may occur, but only rarely. The merchants are often companies that are motivated to check their account balance frequently, while the individual buyers are offline most of the time.

- Payment processing platforms like PayPal/Alipay

. Users of such platforms typically leave some money on the platforms and use them to buy product/services from merchants later. Payment only flow from the individual consumer group to the merchant group.

Figure 1

. Payment sender/recipient group and the token flows

Obviously there are many more real world applications like these, but we’d omit them here. In such cases, users in the recipient group are typically larger organizations/companies, and can reasonably be assumed to be online often. They can do frequent balance check and withdraw their balance if they find the Plasma operator cheating. However, users in the sender group are typically individual consumers that are offline most of the time. Hence, for this group of users, it would be beneficial if the Plasma operator can offer some guarantee that their tokens in the Plasma are safe even when they are not actively checking.

To achieve this, we propose the following off-chain state verification protocol. The Plasma operator maintains a state, which is an augmented Merkle tree containing the token balance for each user. The operator needs to periodically commit the root node of the state Merkle tree to the main-chain. On the other hand, the on-chain Plasma smart contract periodically samples and validates a small random subset of the state Merkle tree. We will show that with our proposed construct, if the Plasma operator cheats, there will always be some inconsistency in the Plasma state Merkle tree

, which can be detected by the random sampling process with arbitrarily high probability. STARK techniques can be employed to improve the efficiency. Further, we require the operator to deposit a certain amount of collateral to the on-chain smart contract, if any inconsistency is detected by the smart contract, part or all of the collateral will be slashed. Hence the proposed mechanism could effectively discourage the Plasma operator from cheating.

Plasma State Construct

The proposed Plasma construct consists of a Plasma smart contract deployed on the main chain, and an account-based Plasma off-chain state.

The account-based Plasma off-chain state is stored in an augmented Merkle tree, where the leaf nodes represents the accounts. Each node in the state tree is augmented with a number bal

. For a leave node, bal

is equal to the token balance of the corresponding account. For an intermediate node, its bal

is equal to the sum of the bal

of its two child nodes. We would call this the summation property

. Specifically, the bal

value of the root node equals to the total number of tokens deposited into the Plasma system. Such node augmentation is similar to the Liquidity Network.

A new leaf node (i.e., an off-chain account) can be created by an on-chain deposit transaction through the smart contract. Furthermore, each token deposit into an existing account also needs to go through the smart contract with an incrementing sequence number seq

. For example, the sequence number for the account creation deposit transaction is 1, and the second deposit to the same account has sequence 2, etc. The sequence number seq needs to be included in the leaf node along with bal

.

We denote this off-chain state S

. In addition, we require the Plasma operator to store the history of S

, similar to how Ethereum stores its states in the blockchain. We denote the offchain state at main-chain block height h as Sh

. The Merkle root needs to be written to the main-chain periodically. For example, whenever a new block is generated on the main-chain. The figure below depicts the off-chain state Merkle tree at a certain height.

[

plasma_state_merkle_tree

1193×605 27.4 KB

](https://ethresear.ch/uploads/default/original/2X/7/7594bb98e6c7833c3311d021464a5aeb6c1dd2a1.PNG)

Figure 2

. Merkle tree for the off-chain Plasma state

Assume user A is in the sender group, and user B is in the recipient group. To send tokens through Plasma, besides the usual transaction signing, we also require that both A and B sign their corresponding leaf node with the updated balance bal

, along with the current sequence number seq

, and its address addr

:

sign

(ski

, addri

|| seqi

|| bali

)

This signature also needs to be stored in the leaf node, as shown in the figure above. Both user A and B need to wait for the Plasma operator to return the Merkle branch after the Plasma state has been updated. Next, they need to validate the Merkle branch, and that the Merkle root is the same as the one written to the main-chain.

Probabilistic Plasma State Validation

The on-chain Plasma smart contract is also responsible for validating the Plasma states. To operate a Plasma chain, the operator needs to deposit a certain amount of collateral onto the smart contract. If malicious behavior is detected, part or all of the collateral can be slashed. This can effectively discourage the Plasma operator from cheating.

The smart contract needs to perform a couple checks for each new block generated

on the main-chain. Assume the current block height is h

:

1. The on-chain smart contract checks needs to check the conservativity of the tokens in the Plasma. For this, it verify the following equality: balh

= bal

h

-1

- D

h

-1

-W

h

-1

. Here balh

is the balance of the root node at main-chain height h

, D

h

-1

and W

h

-1

are the total amount of deposit and withdrawal at height h

-1. This check is similar to the Liquidity Network.

1. The smart contract also randomly selects m

Plasma account addresses. And the Plasma operator needs to submit the Merkle branches to these m

accounts for state Sh-k

to the smart contract. Here k

is an positive integer specified by the smart contract (i.e. k

= 1), so that when the m

addresses were selected, the Merkle root of Sh-k

has already been committed to the main-chain. The smart contract then validates these Merkle branches. Here we note that the smart contract is aware of all the Plasma addresses, since every Plasma address was created through the smart contract.

1. In the meanwhile, as we assume earlier, users in the recipient group are often online and check their accounts balance and the Merkle branch for their accounts periodically.

To see why the process described above can detect cheating attempts from the Plasma operator, we can prove the following invariant.

Invariant

: If the Plasma operator cheats, there is always at least one inconsistency in the state Merkle tree, unless the operator reverts the changes

.

To prove this invariant, we note that the Plasma operator only has the following ways to steal token from legit user accounts:

The Plasma operator creates a new account (i.e. a leaf node in the Merkle tree) with arbitrary amount of tokens, or increase the token amount in his existing accounts. However, it is easy to show that this will either violate the summation property mentioned earlier, or violate the token conservativity check balh

= bal

h

-1

- D

h

-1

-W

h

-1

. The reasoning is similar to the Liquidity Network.

Based on the reasoning above, the Plasma operator cannot create tokens out of thin air. Hence he has to steal from other accounts. One possibility is that the Plasma operator could try to decrease the balance of a legit account by x

tokens, and then increase one of his account by x

tokens. If the victim account is an online recipient user, he will soon find out about it. If the victim account is an offline sender address, this operation also introduces an inconsistency. As mentioned earlier, the leaf node contains the signature of the account holder:

sign

(ski

, addri

|| seqi

|| bali

)

Furthermore, we note that for a sender account, its balance bali

for the same sequence number never increases

. This is because that sender's account balance increases only when there is a deposit through the smart contract. Each such deposit increments the sequence number, which the smart contract is also aware of. As a result, it is not possible for the Plasma operator to forge a valid signature for bali

- x

tokens, since the account holder has never signed such balance with the current sequence number before.

Then, the last option for the Plasma operator is to replace the address of a legit account with his own address, and sign the leaf node with his private key. And then withdraw the tokens from the replaced account. However, this also introduces an inconsistency. If the replaced account is an online recipient user, he will soon find out about it and submit the evidence to the smart contract. If the replaced account is an offline sender user, such replacement would lead to an inconsistency between the smart contract and the state Merkle tree – an address recorded in the smart contract is now missing from the Merkle tree.

In each of the above scenarios, at least one inconsistency is introduced into the state Merkle tree. And the inconsistency will persist in the tree (although the operator could change the location of the inconsistency for each height), unless the Plasma operator reverts the changes.

Analysis

Now consider a case where the Plasma operator stole some tokens from a users and issued a withdrawal request to the on-chain smart contract. This introduces at least one inconsistency somewhere in the state Merkle tree. The location of this consistency in the tree might change, but it cannot be eliminated. Similar to the Plasma MVP, the withdrawal request needs to go through a challenge period, say t blocks.

During this period, for each new block, the smart contract samples m

leaf nodes and the corresponding Merkle branch, and assume that there are n Plasma accounts, this can at least has m / n

probability to detect the inconsistency. Repeating such sampling for t

times, the probability of successful detection is

$$1 - (1 - m / n)^t$$

Yet this still requires the on-chain smart contract to perform significant amount of computation, if m

is large enough. To reduce the load of the main chain, we can leverage the STARK mechanism

. The time complexity of generating STARK proof for the Merkle branches for m leaf nodes is

$$O(m \log(n) \log(m \log(n)))$$

The amount of data the Plasma operator needs to submit to the on-chain smart contract is just a constant. And the time complexity for the smart contract to verify the proof is

$$O(\log^2(m \log(n)))$$

We can select m

properly such that the $O(m \log(n) \log(m \log(n)))$ proof generation time is reasonable for the Plasma operator. One extreme is $m = n$

, however, when n is large, it could be too much computation overhead for the Plasma operator to generate the STARK proof for the entire tree. We can choose m to be a relatively small number. To give an example, say n

$= 1{,}000{,}000$, and m

$= 500$, if the challenge period t

> 9400 blocks (about 26 hours if the block time is around 10 seconds), the probability of successful detection one inconsistency is over 99%. Moreover, n

$= 1{,}000{,}000$ and m

$= 500$ should be relatively small for the STARK proof generation task.

We also note that the above analysis assumes the Plasma operator only introduces one consistency to the state Merkle tree. If the Plasma operator tries to steal from multiple accounts and introduces multiple inconsistencies, the detection probability will be even higher. For example, say the operator tries to steal from 1% of all the accounts, the detection probability is roughly

$1 - 0.99mt$

Even with a small m

$= 10$, after t

$= 50$ blocks (less than 10 minutes), the inconsistency will be detected with more than 99% probability.

Further Discussions

There are other ways a malicious Plasma operator could attack the system. One potential attack is a sybil attack, where the Plasma operator generates many accounts to make the inconsistency detection more difficult. To handle such attacks, the smart contract could require the operator to sample a fixed percentage of accounts (e.g. 0.5% as in the example above).

Moreover, we note that although most of the token flows from the sender group to the recipient group, the reversed token flows are also possible (e.g. refund). However, in real world applications, the reversed token flows are much rare, so such transactions can be performed on-chain.

Finally, we want to note that the proposed off-chain state validation framework can work in conjunction with the exit game mechanism. It just further enhances the security of the system.