Plasma Cash takes the promise of Plasma - an Ethereum scaling solution - one step further.

Plasma promises a way to scale blockchain technology by reducing the number of transactions Ethereum needs to process. By distributing this burden across a theoretically limitless network of “sidechains,” the Plasma implementation vastly increases blockchain’s transactions per second while dramatically lowering costs from transaction fees. It also provides a secure way to withdraw funds if ever a sidechain were to become compromised.

Plasma Cash builds upon this foundation, but requires that users watch only their assets instead of keeping track of everything. That, among a host of other benefits, help scale blockchain even further.

Karl Floersch writes a great article about his first specification of Plasma Cashthat covers the various benefits of Plasma Cash, how the specification should work, and what cases you need to cover in order to provide security guarantees. At first glance it looks quite simple, but diving into the details reveals a few things that make the implementation not straightforward.

At Lucidity, we wanted to create our own implementation of Plasma Cash that would be much simpler. Today, I’m proud to share the work of two members of my team - Dariusz Zacharczuk and Alex Voloshko - who spearheaded the development of Lucidity’s new Plasma Cash implementation - the simplest and most efficient implementation to date.

GitHub

luciditytech/lucidity-plasma-cash

Lucidity's Plasma Cash Implementation. Contribute to luciditytech/lucidity-plasma-cash development by creating an account on GitHub.

Deposit

Deposits are simple. A user makes a deposit to a Plasma contract in order to go off-chain. Once the deposit is made, a unique depositId

identifier is issued.

Sidechain Transactions

Once you have your deposit, you can start transacting on the sidechain. Each transaction contains:

uint256 depositId; uint256 prevTxBlockIndex; address newOwner; uint256 targetBlock;

The most important part here is that we are specifying the previous block index that contains input transaction for the current one: prevTxBlockIndex

.

Using targetBlock

as one of transaction parameters, will prevent from stealing the money for all cases, where operator should include user block, but he did not ie. withheld the off-chain Tx in order to make an attack for deposit. Not placing off-chain transaction on targetBlock

position automatically makes that transaction invalid.

User can start exit on that transaction, but then the right owner will start an exit on previous one and he has the priority for exit. It is not possible to challenge right owner with invalid transaction, because plasma root for targetBlock

will not match.

These attributes are enough to provide security guarantees and validate transaction history.

Based on these attributes, a hash (leaf for merkle tree) is generated and must be signed by the sender. Plasma operator includes this transaction in the mempool, (re)creates the merkle tree, and submits a block to the plasma contract with the merkle root.

Plasma users need to keep track of all the transactions for depositId

that they currently own. These transactions can be validated so each user can be sure that the sidechain is operating as it should. In the case it isn’t, users can start an exit procedure. Transactions can also be used for challenging invalid exits.

Start Exits

When a user notices a problem on the sidechain or simply wants to withdraw their money, they initiate an exit. There are

two kind of exits:

1. deposit exits
2. transaction exits

The difference is that for the first case thee user need to provide a depositId

and for the second case a transaction

needs to be provided, based on which deposit the user owns.

Here you can see how plasma contract stores the exit:

struct Exit { address exitor; uint256 finalAt; bool invalid; } /// @dev depositId => blockIndex => Exit mapping(uint256 => mapping(uint256 => Exit)) public exits;

Each exit has the following information:

- whether it's valid or not (has been challenged);
- time when it can be finalized;
- the exitor.

But the most important part is how

the exits are stored. They are stored as a matrix

of depositId

s and blockIndex

es. This is important for a number of reasons:

1. The matrix allows users to store any number of exits for any number of deposits;
2. It guarantees that there is only one slot where a particular exit can be stored;
3. Exits are kept in correct order - from the oldest to the newest (blockIndex

works like a timeline for the deposit history);

1. Each deposit history is stored independently.

Exit Queues

// @dev depositId => exitQueue mapping(uint256 => uint256[]) public exitQueue;

When an exit is submitted, we need to know which slots in our matrix are taken. This is what the queue is for.

We have an individual queue for each deposit. Each queue keeps all block indexes for which exits were made and it keeps them in the correct order. Order is based on blockIndex

, so it's based on the history of transactions, not on the time of exit creation - this is critical for providing safety.

Challenges

In case a user tries to exit the plasma chain with somebody else's money, there is a challenge period to prevent the exit. We specify a simple challenge mechanism where other users can provide a proof that the exit has been spent. This is where the ordering provided by prevTxBlockIndex

is useful because it prevents someone from challenging the exit with an older spend transaction.

Let's look at a few scenarios:

1. A user wants to exit on deposit
2. If NO sidechain transaction have been made then nobody can challenge that exit and the user can have his/her money back.
3. If the deposit was transferred to someone else then receiver of the transferred deposit needs to challenge that exit

with the spend transaction

4. When a user wants to exit from a valid block produced by the plasma operator(s)
5. If the exitor is not attempting fraud and is exiting with the last transaction on the deposit's history, then nobody can challenge that exit.
6. If the exitor is attempting fraud, with not the last transaction, then anybody can challenge the exit.
7. In all other scenarios there is a possibility to exit with the latest valid transaction, because there is no subsequent spend.

Finalize Mechanism

After the challenge period, the exitor can finalize the exit request. When the finalize function is called, all exits that have passed the challenge period will be processed in order based on their blockIndex

. When the qualifying exits are processed the prioritized exit succeeds and the exitor balance is updated with the deposit amount and the bond, and all other exits will be invalid.

Each depositId

has a dedicated priority queue. Exits are ordered by blockIndex

within a queue.

Individual exits also have massive impact on performance. The contract processes exits only on a particular deposit in one call. In most cases, it will be only one, so the exitor will pay only for its exit.

What makes it different?

We want to highlight the main differences between Karl's specification and ours:

- only one transaction is required per exit and challenge;
- there's no need to reply to challenges;
- as a result, it requires less gas to operate.

These enhancements are possible because of more convenient data structures to handle exits.

Future improvements

Below is a set of improvements that we plan on implementing:

- At the moment, we are using 256-bit numbers to encode depositId

and that requires 256 elements in Merkle proofs. We are looking forward to optimizing Merkle Proofs for Sparse Merkle Trees;

- Add ERC20 and ERC721 support ;
- Use RSA Accumulators for Plasma Cash history reduction
- Utilize defragmentation techniques.

Go

Plasma

!

GitHub

lucidity tech/lucidity-plasma-cash

Lucidity's Plasma Cash Implementation. Contribute to luciditytech/lucidity-plasma-cash development by creating an account on GitHub.