Loom Network: Plasma Cash for ERC721 Tokens

Hello everyone,

I'd like to announce the Plasma Cash implementation that we have been working on at Loom Network for the last 3 months. It is a Plasma Cash implementation specialized for ERC721 tokens, which means that token splitting/merging is not a requirement due to ERC721 tokens being non-fungible and unique on their own.

In the rest of the post I'll explain our implementation, parts of existing research that were reused, the advantages of this construction as well as some disadvantages.

For following along with the code, you can checkhttps://github.com/loomnetwork/plasma-erc721/

For less-technical readers you can read our blog post:https://medium.com/loom-network/plasma-cash-initial-release-plasma-backed-nfts-now-available-on-loom-network-sidechains-37976d0cfccd

Plasma Contract

Plasma Cash as an ERC721 Receiver

A smart contract can execute some functionality in response to receiving ether by defining a payable fallback function. This does not apply when receiving any kind of tokens. The ERC721 standard, defines a onERC721Received

method which is called when an ERC721 token is sent to smart contract account, and can be used to mimic the functionality of a fallback function as a callback after receiving ERC721 tokens. We implement our Plasma contract as an ERC721TokenReceiver

, which triggers a deposit function right after receiving a coin.

Sparse Merkle Trees

Each coin in Plasma Cash is unique, and considering that we use Merkle Trees to organize the transactions inside a block, we use a Merkle Tree where the index of a transaction is the uid of the coin. Hereafter, this shall be referred to as a "slot" in a Merkle Tree. Furthermore, we can assume that the slot is generated in a random way (e.g. through a hash function on unique inputs), and due to the reasonable assumption that not all coins will move in 1 block, that Merkle Tree is expected to be sparse. Utilizing the technique from a previous research post we create a Sparse Merkle Tree (SMT) implementation in Javascript and Golang, and by modifying OmiseGO's implementation in Python as well. The Solidity SMT verifier code was found in etherscan written by the authors of the previously referenced research post. This results in being able to have compact proofs of inclusion and exclusion (a proof of exclusion for a slot is a merkle branch that proves there is an empty hash at the slot in the SMT)

Transaction format

A Plasma Cash transaction is a tuple of: (slot, previousBlock, denomination, owner, hash). When a transaction's previousBlock is 0, it is a deposit "out of nowhere" transaction which is the transaction that gets created right after a coin's deposit. We differentiate the hashes of deposit transactions to be the keccak256 hash of the coin's slot, in order to have an easier verification process. The denomination of a transaction is currently always set to 1 and will be adjustable in the future as we're exploring splits/merges or further investigating the possible integration of Plasma Debit.

Deposits

A deposit gets initiated only after the Plasma contract receives an ERC721 token via the onERC721Received

function. This creates a new block on the Plasma contract containing only the deposit transaction for the newly deposited coin, and emits an event that notifies the Plasma chain about the coin's deposit.

Coin State - Exits - Challenges

A coin has a uid

field which is the uid of the ERC721 coin when it got deposited. It is allocated by the ERC721 token contract and is not related to the previously referenced slot

. A denomination field which is currently set to 1 always is used to track the denomination of the coin, this is related to porting the implementation to support Ether and ERC20 contracts, which essentially would have a null uid field and a denomination

field >1. The contractAddress

field is added to allow the contract to support multiple tokens.

We model the coin’s state as a state machine, where the states are DEPOSITED

, EXITING

, CHALLENGED

, EXITED

.

A freshly deposited coin is initially in the DEPOSITED state. A user can initiate the exit of a coin by referencing a transaction that they own, as well as a direct ancestor of that transaction, along with their corresponding included blocks. After starting the exit of a coin, it transitions to the EXITING state.

In the “happy case”, a coin that has been in the EXITING state for a certain time period (challenge/maturity period) after a successful exit can have its exit finalized and transition to the EXITED state where it can finally be withdrawn, as shown in the figure below.

[

exit

873×415 49.4 KB

](https://ethresear.ch/uploads/default/original/2X/8/85dab5c31999bb849502454c7cbccf0ab7a33c43.jpg)

In the “unhappy-case”, a challenge occurs during the maturity period of a coin’s exit. As described in Karl’s Plasma Cash spec, in the case of a challenge with a later transaction or a transaction between the exit’s referenced blocks, the exit gets instantly deleted; and the bonds of the exitor get given over to the challenger. In the case that an exit is challenged with a transaction at an earlier block, the coin’s state gets set to CHALLENGED and the challenge must be responded to by the end of the challenge period. If the challenge is responded to, the coin’s state returns to EXITING, and the exit will be finalized as in the cooperative case; otherwise, the exit gets deleted.

The following limitations apply:

- An exit can only be initiated by the exiting transaction’s owner.
- A challengeAfter can only be initated with a DIRECT SPEND of the exiting transaction of the challenged exit.
- A challengeBetween must provide a spend from the owner of the previous transaction of the challenged exit.
- A challengeBefore is a user attesting that they can exit the coin, but instead of exiting it they challenge a malicious exit
- A respondChallengeBefore must be with a later spend of the transaction of the challenge being responded.

Regarding bonds, we require that an exitor submits a 0.1 ether (can be adjusted) bond when they initiate an exit. If there is a successful challenge their bond gets slashed over to the challenger. challengeAfter

and challengeBetween

do not require bonds as they are non-interactive, however challengeBefore

requires that the user submits a bond for the challenge, which gets slashed if there is a successful response.

Validators

We implement a minimal Validator Manager Contract which allows the owner to authorize other addresses to perform validation. This is done to separate the governance of the Plasma Chain-Contract in a way that can be customized by the operators of the chain.

Conclusions

Robustness / Security

We tested for a variety of possible scenarios of exits/challenges as well as cases where there are invalid/withheld blocks, currently the implementation is robust and there is no case where an invalid exit cannot be challenged and result in theft of funds.

Benchmarks

In order to simulate scenarios outside truffle tests, we cloned theOmiseGO Plasma Cash repository and modified their python child chain implementation to support our setup.

We describe the currently implemented benchmarkhere. Our finds are the following (we consider 102569 gas for each safeTransferFrom

call to an ERC721 contract):

- With each block having an approximate gas limit of 8 million, this means that at best we can move 78 ERC721 tokens per block which is ~6.4 transactions per second. With Plasma, we can effectively compress the contents of a Plasma Block in 1 root and allow for any number of coins to change ownership in 1 Root chain block (limited by the size of the Plasma Block, which we can assume is large). In our benchmark, we have no issue supporting 2000 coins changing ownership per Root chain submitted block. Note that 2000 transactions in a block that can potentially handle 2^64 transactions is a very small percentage.

- The gas costs for a full exit are the sum of depositing, starting and finalizing the exit and withdrawing the coin. This totals to ~800k gas required, which means that we'd need a coin to change ownership more than 8 times, in order to fully amortize the gas costs from being in the Plasma chain. We consider this a reasonable amount of transactions for users that will be actively utilizing a Plasma chain's benefits.

- The operator spends ~74k gas per block submission, this should be compensated via fees, and needs to store ~134 bytes per transaction, which is approximately 261Kb for blocks with 2000 transactions. The requirements for running a plasma chain operator naturally depend on the frequency that blocks get generated and submitted to the rootchain as well as the size of the blocks.

Tradeoffs

Pros:

- Due to each coin being unique in Plasma Cash, there is no need for an exit priority queue, as in MVP constructions. Each coin is unique and its exit can be challenged separately. For the same reasons, an operator cannot create a large transaction which steals all coins and thus mass exits are not required for security (although they can be used for gas optimizations)

- Exit games and challenges are simple

Cons:

- Non-fungibility = Cannot split a coin. This is currently not an issue, however expanding Plasma Cash to be more flexible and operate with ETH/ERC20 which can be split would be great, Plasma Debit can help with that.

- Linearly increasing size of proofs, can be mitigated via Plasma XT.

Future Work

- Adapt to ERC20/ETH (trivial)

- Investigate possible Plasma XT / Plasma Debit implementations

- Investigate further standards that we can support (e.g. ERC998, ERC1155 for collectibles)

- Investigate solving griefing attacks through limbo exits (operator forces a user to pay the bond amount in order to complete a withheld transaction)

- Investigate submitting multiple Plasma blocks in 1 root chain block to allow even more throughput. This maybe has finality issues which are not clear at the moment.

You can clone our repository athttps://github.com/loomnetwork/plasma-erc721.

Thank you for reading!