This covers what we have worked out in our Plasma Cash implementation for these implementation choices:

- Sparse Merkle Trees (SMTs) for compact proof of spend (and in some cases, non-spend), where each Plasma block has a Merkle root published on MainNet representing the spend of a tokenID in that specific block.
- Bloom filters, for compact proof of non

-spend

- Probabilistic transfers

Feedback to improve our & everyone else's Plasma Cash implementations appreciated!

Sparse Merkle Trees

Sparse Merkle Trees are best suited for Plasma Cash implementations over more familiar Binary Merkle Trees (BMTs) because:

1. when a token is spent, you can guarantee

that the tokenID has exactly one transaction in the block, which you don't get with BMTs.

1. when the token is NOT spent in the block, you can also prove it with the same mechanism. This will come in handy when Bloom filters generate false positives.

The central idea behind SMTs representing which tokenIDs have been spent in a block is you can keep transaction hashes of the n

tokens spent in the block at 2^q leaves, where each tokenID is a q-bit number. Because n << 2^q, and with random tokenIDs made truly sparse with a hash function (e.g Keccak256), there is either a "null" or the transaction hash at each leaf.

For SMTs to be built in practice, just the Merkle branches for the n

Merkle branches need to be computed. It is valuable to pre-compute a set of default hashes

for all q levels, following Laurie and Kasper (2012) and

Dahlberg et al (2016):

- At level 0, the default hash is H(0)
- At level 1, the default hash is H(H(0),H(0))
- At level 2, the default hash is H(H(H(0),H(0)),H(H(0),H(0)))
- … and so on

A typical Merkle proof for a 64-bit tokenID is NOT 64 32-byte hashes going all the way from level 0 at the leaf all the way to the root! Instead, q

bits of proofBits

(with q=64, uint64

) can compactly represent whether the sisters going up to the root are default hashes or not – that is, for each bit:

- 0

means "use the default hash"

- 1

means "use 32

bytes in proofBytes

"

where the proofBytes

contains just the non-default hashes. At the leaf of the Merkle branch it is natural to keep the 32-byte RLP hash of the transaction bytes.

A checkMembership(bytes32 leaf, bytes32 root, uint64 tokenID, uint64 proofBits, bytes proofBytes)

helper function in Ethereum MainNet can take proofBits

& proofBytes

and prove that a exit or challenge is valid if it matches the Merkle roots provided by the Plasma operator in submitBlock(bytes32 root)

; likewise, one user receiving a token from another user must [without any checkpoint finalization concepts] get tokenID, along with t

raw txbytes

, and t

Merkle proofs, where each Merkle proof concerns the spend in a specific block. But note that a non

-spend also can be proven as well, where the leaf is H(0)

.

In the best case, a single transaction in a block represents the spend of just 1

token (n=1

), and instead of a 64 x 32

byte proof, you have default hashes all the way from level 0 to level 63, and proofBits

is 64 zeros (0x0000000000000000

). You have proofBytes

being nothing and a uint64 being 0

, which is as compact as it gets! Proof size: 8 bytes.

In the next best case, if you have 2 tokenIDs (say, 0x01234...

and 0x89abc...

) the proof of spend of each token would have one non-default hash up at top in level 63 and proofBits

being 1 followed by 63 zeros (0x8000000000000000

). Proof size: 40 bytes.

For non-contrived cases, SMTs will have very dense upper nodes from level q-1

down to around level log_2(n)

. To make this concrete, lets say you have 10MM Plasma 64-bit tokens, where each token undergoes 500

transactions per token per year. Then you'll have 5B transactions for your 10MM tokens each year. If your Plasma block frequency is 15s/block, then you'll have these 5B transactions distributed over 2.1MM blocks/yr, with an average of 2,378 transactions per Plasma block (500 x 10 x 10^6/(86400 x 365 / 15)

). When you put these 2,378 transactions into an SMT, because log_2(2378)=11.2, you'll have a super dense set of nodes mostly having non-default hashes for levels 63 down to level 53 or so, and then below that you will have just one tokenID going all the way down to level 0. Proof size: 32 bytes * 10 levels, or 320 bytes.

We decided to do 64-bit tokenIDs instead of 256-bit tokenIDs because token collisions are still astronomically unlikely and:

- the proofBits

are 24 bytes smaller (uint64 instead of uint256)

- less gas is spent in checkMembership

on all 0

bits in proofBits

- we can have a smaller 64-element array of default hashes

computed instead of 256 hashes. Less hashing, means less gas and happier users.

Using the same logic, it may be natural to use uint32

(4 bytes) to save on gas a bit more, provided that collisions between circulating tokenIds verified to be impossible upon deposit

events.

You can combine the fixed length proofBits

and variable length proofBytes

into a single proof

bytes input for exits, i.e.

startExit(uint64 tokenID, bytes txBytes1, bytes txBytes2, bytes proof1, bytes proof2, int blk1, int blk2)

The analogous challenge interfaces will then have fewer argument inputs in the same way.

Bloom Filters

It is not

enough for a user receiving a tokenID from another user to just validate t

Merkle branches for the t

transfers. It is also necessary for a user to validate that the tokenID has NOT been transferred in between each of those t

transfers. To help prove this non

-transfer, the Plasma operator can be required to publish a Bloom filter compactly summarizing the tokenIDs spent in each block. Then every user can verify non-transfer for any candidate tokenID being transferred to them.

The validation of whether the Bloom filter published by the Plasma operator is correct is really easy: for all the transactions included in the block, the Bloom filter must respond Yes, in block

. Having a false positive (a transaction that is not in the block also firing Yes, in block

) is normal, but if the Plasma operator publishes a Bloom filter where a transaction included in the block does not

get reflected in the Bloom filter output, the user should exit.

How expensive is this, in terms of real storage somewhere

? Taking the 10MM token example with 2,378 transactions per block:

- 8K Bloom filter: an awesome 1 in 499K
- 4K Bloom filter: a tolerable 1 in 407

For a whole years worth of storage of 4-8K Bloom filters, you'll need 8-16G ((86,400 x 365/15) x 8,192/2^30). Obviously, it is infeasible to store these large raw Bloom filters on MainNet. It then becomes natural to publish 32-byte hashes of the Bloom filters alongside Plasma block Merkle roots so that the interface for submitting new blocks on MainNet becomes

submitBlock(bytes32 merkleRoot, bytes32 bloomFilterHash)

Any user can download a Bloom filter from the Plasma operator by the published bloomFilterHash

, and verify it against the broadcasted blocks:

- for any actual spends of the users token, the Bloom filter must say "1". If it doesn't, the user exits.
- for any non-spends of a token, the Bloom filter might say "1" (with something like 1/407 or 1/499K chance), in which case, the user must extend its proof of token ownership slightly to record this false positive.

The Plasma operator can expose a verifiable getBloomFilter(bytes32 bloomFilterHash)

that returns the raw data of the Bloom filter. The raw data for the Bloom filter must be verified to hash to bloomFilterHash

. If the Plasma operator publishes a checkpoint every 1-2 weeks that can be considered finalized, the 16GB of storage per year goes down to 315-630MB per week. On the other hand, with a much higher turnover than 500 transactions per tokenID per year, or a lot more tokens than 10MM, you'll need more than 8K chunks, but Bloom filters scale linearly. If you have 10x the number of transactions per block, you need 10x more storage for your Bloom filters to have the same false positive rate.

Probabilistic Transfers and Receipts

Plasma Cash's core limitation of requiring non-fungible

tokens may be solved with an adequate spec of merges and splits, but in the meantime, a transaction can include a probability of transfer p

, hinted by Karl's Plasma Cash Simple Spec. If the owner of a a token with denomination 10 can submit a transaction with "p=.5" with the token transfer, then the receiver can consider this the same as receiving a token with denomination 5, even though no such token exists. This approach may be less expensive than an approach that supports splitting the denomination 10 token into 2 tokens each with denomination 5.

Generally, if the Plasma operator can guarantee adequate RNG, then a token transfer for denomination D

with probability of transfer p

can be considered the same as a token transfer for denomination p

x D

with 100% chance of success.

- Adequate RNG must be provided with a single RANDAO transparent external source that publishes a random number r

at the same frequency of the Plasma blocks themselves. The Plasma operator cannot

be in control of the RNG because then it may collude with the sender or recipient in such as a way so as to favor the sender or recipient. If p

is represented by P bits, then the lower P bits of r

must be less than p

for the token transfer to succeed. It is then necessary that the transaction include the transfer target_block

so that the Plasma operator cannot collude with the sender or recipient to prevent the recipient from getting the potential token.

- Successful (or unsuccessful) transfers will have receipt 1

(or 0

), and this receipt value is contained in txbytes

of leaves in the published SMT root. The SMT will include transactions with receipt 0

and of receipt 1

. This way, the recipient of the token can be provided by the sender the Merkle branch proof that an attempt to provide p

x D

of value, and the recipient verifies whether the attempt to transfer the token was really made, and specifically will check if the RNG process is "true" RNG.

- Having Bloom filters model unsuccessful transfers (with receipt 0

) does not help new owners verify ownership history. So, published Bloom filters must only consider the successful transactions, marked with receipt 1

in the SMT. Bloom filters for Plasma cash help proofs of non-transfer rather than non-spend.

- Similarly, when exiting using the last 2 transactions, only the last two successful

Merkle proofs should be permitted. Because the txbytes

of the last 2 transactions contains the receipts, the requirement in checking valid submissions for exits and challenges will be that receipts are 1

in all cases.

Notes:

- Because the SMT includes the unsuccessful transfers, a minimum value of p

should be established so as to limit the size of proofBits

and not have frivolous numbers of transactions tiny p

dominate proof computation, storage and transmission.

- An alternate design can be considered where there are two SMTs published: one for the successful transfers with receipt 1

and one for unsuccessful transfers (with receipt 0

); but this design loses the important property behind the single SMT that there is necessarily only one transaction possible for the tokenID.