Prerequisite: RSA Accumulators for Plasma Cash history reduction

In a naive RSA Plasma Cash implementation, each coin is assigned a unique prime number as a coin ID, and the RSA accumulator update for that block includes the coin ID for each coin that was spent. For a user with a width-n

fragment, a proof of inclusion has size O(N)

and a proof of exclusion has size O(N)

, and the operator has O(N)

overhead, making this scheme ineffective if we want to achieve very fine denominations.

We'll start off with a scheme that cuts down proofs of exclusion to size O(log(N))

at the cost of increasing proofs of inclusion for one coin to O(log(N))

(a proof for N

coins only increases by a factor of two). We generate a tree of primes associated with indices of coins and subsets of indices corresponding to nodes in a binary tree, for example for 4 coins as follows:

To include coin 7, you would set A' = A^{2 * 3 * 7}

. To include both coins 13 and 17, you would set A' = A^{2 * 5 * 13 * 17}

, to include both 11 and 13 you would set A' = A^{2 * 3 * 5 * 11 * 13}

. To prove membership of eg. coin 17, you would need to prove that 2 * 5 * 17

is part of the accumulator (ie. A'

is a known power of A^{2 * 5 * 17}

).

To prove non-membership of an aligned slice

(a slice which corresponds exactly to a subtree), only a single proof of non-membership, eg. of 3

, is required. For arbitrary slices, you can construct a proof that batches together log(n)

proofs of subtrees; in general, any slice [a,b]

can be decomposed into log(b-a)

adjacent aligned slices. For example, to prove that 11, 13 and 17 are all not part of the accumulator, you would need to prove that 11 and 5 are both excluded, and so you need simply prove that log_A(A') \mod 55

is a value that is not a multiple of 5 or 11. Hence, proving non-membership of a range can now be done compactly. However, proving membership of a range unfortunately still requires a batched proof of all of the individual coins.

We can deal with this problem by extending the scheme further, making a Merkle forest instead of a tree:

To prove membership of a single aligned slice, you now need to prove several things:

- The leaf value corresponding to the subtree is included, as are all of its ancestors up to that root.
- The leaf values in any higher tree in the forest are not included.
- The value corresponding to the subtree is not included in any lower tree in the forest.

This is best illustrated by example. Suppose we want to prove membership of coin 1 (ie. the second coin from the left). Values for which we prove membership are colored greed, and values for which we prove non-membership are colored red:

Now, suppose we want to prove inclusion of the aligned slice containing coin 0 and coin 1:

The general principle is that there are separate trees that are used to prove membership of an aligned slice at each 2^k

size level, and this proof of membership includes a proof that aligned slices intersecting with that slice at higher or lower levels are not included.

Now, to prove that a given aligned slice is not included, we make a proof as follows (using coins 0 and 1 as an example):

This proves that:

1. The aligned slice [0...3]

cannot be included (as that would require including the prime 2

, which we prove is excluded)

1. The aligned slice [0...1]

cannot be included (as that would require including the prime7)

1. The individual coins 0

or 1

cannot be included (as either of those proofs would require including the prime 13

)

All of these proofs are of size log(n)

for n

total coins, and this includes both cost of verification and transmission and

cost of construction. If all transactions ever only touched two-coin aligned slices, no one will ever need to do any calculations using the primes 19

, 23

, 29

or 31

.

If a Plasma Cash implementation wishes to use a very large number of coins (eg. 2^{50}

coins), then this is a very helpful feature. However, how do we assign primes to that many coins? Ideally, we want primes that are as small as possible, as these are more efficient to generate proofs for.

One solution is to come up with a function that deterministically generates a prime on-demand for any given index. Given what we know about prime gaps, simply mapping x

to the first prime above 25000 * x

should be sufficient. The problem is that deterministic 100% effective primality tests have a high runtime, making them difficult to use on-chain. A simpler approach (thanks @ldct) is to simply do one round of precalculation which generates the list of the first 2^{40}

prime numbers and stores them in a Merkle tree; any use of the primes on-chain would need to be combined with a Merkle branch. Any client could check the precalculation locally.