The main weakness of Plasma Cash is the indivisible nature of the tokens. To make the system worthwhile to use, the transaction fee needed to exit should be much smaller than the value exited; because in Plasma Cash each unit of value needs to be exited separately, this effectively means that the minimum denomination size in Plasma Cash would need to be several hundred times a blockchain transaction fee (ie. in the dollars).

We can solve this problem by making the denomination very small, and allowing users to “multi-exit” a subtree of coins that all have the same owner with a single Merkle branch. Given a 2^k

sized subtree where every element is identical, a single branch of that tree is enough to provide the information needed to verify the entire subtree and sign off on behalf of it. However, using this in practice leads to a fragmentation issue, as users send each other partial payments and keep breaking their coins up into more and more pieces; after enough payments, a user would have to submit a separate Merkle branch and pay a full transaction fee for each coin that they exit, nullifying the benefit of exiting subtrees.

One solution to this is ongoing defragmentation, and the scheme I will propose is as follows: as a condition of accepting a transaction from A sending coins to B, A and B would have to send each other coins in such a way that at the end, every coin held by A has a lower index than every coin held by B. For example, suppose the starting allocation is B A C D A; B B E B F; G A A H I

, and A is sending B two coins. The starting balances are A: 4, B: 4, so the final balances would be A: 2, B: 6, so the final allocation would be A A C D B; B B E B F; G B B H I

, so the transfer would be an atomic swap: A gives B coins 4, 11, 12, and B gives A coin 0.

A simulation shows that this scheme can keep fragmentation very low indefinitely:

```
import random, math

def mk_initial_balances(accts, coins): o = [] for i in range(accts): o.extend([i] * random.randrange((coins - len(o)) * 2 // (accts - i))) o.extend([accts-1] * (coins - len(o))) return o

def fragments(coins): o = 0 for i in range(1, len(coins)): if coins[i] != coins[i-1]: o += 1 return o

def xfer(coins, frm, to, value): coins = coins[::] pos = 0 while pos < len(coins) and value > 0: if coins[pos] == frm: coins[pos] = to value -= 1 pos += 1 return coins

def unscramble(coins, c1, c2): coins = coins[::] k1 = coins.count(c1) pos = 0 while pos < len(coins): if coins[pos] in (c1, c2): coins[pos] = c1 if k1 > 0 else c2 if coins[pos] == c1: k1 -= 1 pos += 1 return coins

def run_with_unscrambling(coins, rounds): for i in range(1000): c1, c2 = sorted([random.randrange(max(coins)+1) for _ in range(2)]) value = int(coins.count(c1) ** random.random()) coins = xfer(coins, c1, c2, value) coins = unscramble(coins, c1, c2) return coins

c = mk_initial_balances(25, 1000) c = run_with_unscrambling(c, 10000) print(fragments(c))
```

This puts a heavy burden on the security of atomic swap protocols: each transaction is now an atomic swap. Here is one way to do this reasonably efficiently in Plasma Cash: Plasma Cash Minimal Atomic Swap.