Plasma Vector [WIP]

Nathan Ginnever

January 7, 2019

Introduction

This work presents an application of cryptographic universal accumulators [5] that was expanded upon in the blockchain setting by Boneh et al [1]. We apply these RSA accumulators in Plasma [2] and show that Vector can scale to high transaction throughput with low overhead for clients (potentially at the expensive of increased prover complexity from the operator or service nodes...TODO). Vectors [] (VC) have the property of binding data to a position in a list and we explore why this is a desired property in the Plasma Cash []. Using VC with 160 bit message spaces, we show how to construct a key-value map in an accumulator that can hold Ethereum public keys. Our contribution is applying VCs to reducing the inclusion and exclusion proof size of Plasma Cash from an $O(n)$ and $O(b \log n)$ respectively, where b is a number of blocks committed and n is the number of coins a user owns, to an $O(1)$ for both. We will see that replacing the merkle tree entirely with a VC accumulator can increase the granularity of Plasma Cash/flow to allow for smaller payments and *fragmentation* no longer is an issue. It is noted that this work relies on a trusted setup for the RSA group order and the adaptive root assumption.

As with previous Plasma designs, this paper will detail the *deposit*, *send*, and *exit* components which follow directly from Cash[]. The Plasma Vector parent contract does not verify the correctness of state transitions, it only provides a ledger to record accumulator commitments. We fall back on the Plasma Cash challenge functions to ensure that invalid transitions cannot exit. In this case, the state transition is simply defined as updating the owner public key of a position(s) in the accumulator. Verifying the correctness of accumulator updates involves inspecting the public key at position x_i in A, the previous committed accumulator value, matches the signature witness and that A' updates the value at x_i to the new public key.

Vector Commitments

A *vector commitment* (VC) [] is a special case of RSA accumulator with the added property of *position binding*. RSA accumulators prove inclusion and exclusion of prime numbers in a set, but do not allow for arbitrary values to be accumulated to preserve the security properties of the RSA group, [] does note that arbitrary values

may work as long as they have a unique prime factorization. VC allow for an M-bit $\{0,1\}^\lambda$ message to be committed to each of the accumulator vector components by assigning prime numbers to the message bits. Committing to a vector allows the accumulator to act like a sparse merkle tree where we can make a statement that for x, a given vector, that for any component index x_i opening a value v that a forger cannot provide a v' s.t. x_i opens to both v and v'. While a merkle tree has a proof size of $O(\lambda \log n)$, where λ is the security parameter of the hash function, to open a value at a position, VC have just constant sized openings. They also carry over the batching properties derived from previous accumulator schemes [ref accumulators] so that Plasma Vector can batch proofs that any number of values are committed to a given index in constant space.

VC Accumulator Construction

We first construct a classic RSA accumulator as described in [CL02 Lip12]. Choose a group G in Z/ZN^* where $N = pq$ and $|N| = (p-1)(q-1)$ is unknown. pq are two 1024 bit prime numbers (p, q). Select a generator $g \in G$ and let the CRS be (N, g).

Setup

For a VC we add three more properties to the setup procedure, the size of the vector n (note this is smaller than requiring the set x for all $x_1...x_n$), a function map to primes H_{prime}, and a message space M (size of each component message x_i in the vector). For Plasma cash, n will be set to the number of lowest denomination coins we would like the vector to represent. I.e. if we would like to have a plasma chain that can *fragment* [cash doc] to a total of 2^{40} owned ranges we would set $n = 2^{40}$ and H_{prime} will need to map $[0, n]$ to a unique prime.

Update

We define now how to update our VC. Lets start with a trivial VC example and assume that the message space of $M = \{0, 1\}$ for simplicity, but we later extend this space to hold enough bits to register a public ethereum key in the accumulator. This will allow the operator to change the ownership of a coin ID without needing to commit to a seperate merkle tree to include this data. Now for a set of m_i messages to be added to A we generate $p_i = H_{prime}(m_i)$ for all $m_i \mathrel{!}= 0$. We can define $A = g^{init}$ and generate an update as $A = A^{\prod_{i=1, m_i=1}^{n} p_i}$. I.e. $g = 3$, $A_0 = g^{3^0 * 5^0 * 7^0}$ Here we have an accumulator that is set up to hold three coins and is empty as they are all raised to the zero power. Since our message space is only $\{0,1\}$, we can only commit the value 1, so to commit a value to the first and third position, we can accumulate 3 and 7 into A as follows. $A_1 = A_0^{3^1 * 5^0 * 7^1}$. Now A_1 contains the value 1 at the first position x_1 in our accumulator, x_2 contains 0 and x_3 contains 1 to build a vector of 1-bits [1,0,1].

Open and Verify

Here, in our example, the verifier can ask the operator to open the committed vector at position 1 that outputs proof π that can verify that the value is 1 at position 1. To do this we generate a membership witness w for p_i in A_1. Proving that m_i is set to 0 involves generating a *exclusion* proof however. Note that when a VC only contains 1-bit messages it takes on the form of classic RSA [] accumulator. I.e. Ask to open position 1. Since $x_1 = 1$ the operator will generate an RSA [] inclusion proof $w = 3 * 7/7 = 3$ and perform a [wes18] PoKE for large cofactors (omitted here). The verifier will compute that the $g^{w*3} = A_1$

Extending to Arbitrary Message Space

Here we detail how to increase the amount of data that our vector components can hold to that of 160 bits for an ethereum public key. To do this, we notice that we now need a unique prime for each bit committed. Formally we need to modify H_{prime} to output λ primes for a given i input. Consider the message space $\{0,1\}^\lambda$ with hash function $H : M \to \{0,1\}^\lambda$. H_{prime} should associate primes $\{p_j : j \in [i\lambda, (i+1)\lambda)$. In our case, $\lambda = 160$ bits. Given a coin vector space of dimension 2^{40}, we would require $160 * 2^{40}$ which is approximately 2^{48} unique primes and still reasonable to generate in the setup phase. On average we will need 80 inclusion proofs and 80 exclusion proofs per coin index opening. This is a pain point as a public key may have many 0 bits. /textitUsing Bezout coefficients [] we could batch exclusion proofs however we consider a novel method of implementing two accumulators, one where accumulating a prime represents 0, and the other represents 1. This way we can reduce to the problem of proving commitment to a public key to only inclusion proofs. Given that each inclusion proof is 2048 bits we can derive a proof that all coins are owned by a user in 4096 bits. [todo verify these numbers, dont think they are correct] I.e.

Have the operator commit to two VC accumulator values A_i and A_e where including a prime in A_i represents committing to the inclusion of a value i.e. a bit is 1 not a 0. And then A_e represents committing to exclusion of a value i.e. a bit is 0 not 1 at a given position i in vector x. In Plasma Vector we want a sparse vector that stores only the public key of the owner of the coin at a particular index. Assume a public key is only 8 bits i.e if we have two values $v_1 = [01101110]$ and $v_2 = [10100111]$ we could generate the two VC accumulators A_i and A_e like so

Let $v_1^p = [3, 5, 7, 11, 13, 17, 19, 23]$
Let $v_2^p = [29, 31, 37, 41, 43, 47, 53, 59]$
Let $A_i = g^{[3^0*5^1*7^1*11^0*13^1*17^1*19^1*23^0]*[29^1*31^0*37^1*41^0*43^0*47^1*53^1*59^1]}$
Let $A_e = g^{[3^1*5^0*7^0*11^1*13^0*17^0*19^0*23^1]*[29^0*31^1*37^0*41^1*43^1*47^0*53^0*59^0]}$

To prove that x_1 opens to v_1 we would ask for a batched inclusion proof for

5,7,13,17,19 to prove that x_1 position contains all of the correct bits with 1. To prove x_1 has all of the correct 0 bits committed we would ask for a batched inclusion proof for 3,11,23 which should result in just two times the batched proof size of one accumulator. Not explicitly proving exclusion could mean that multiple values are stored in the same position since the operator could include more primes than the verifier is checking which will allow other values to pass. To fix this we use [] to compactly prove that A_i and A_e are disjoint accumulator sets.

Proof of Exponent Knowledge PoKE

[todo]

Here we define the protocols that Plasma Vector uses to ensure that the inclusion and exclusion proof size is manageable. Recall [vitalik post], that for a given RSA accumulator $A = g^u$, if we would like to show that an element v is in A the prover must generate a cofactor x s.t. $A = (g^v)^x$. In our case x will be large as it represents all IDs in our coin space, so using it as part of the proof is not feasible. Using an extension of the Wesolowski PoE [] proposed by [bunnz] reduces the burden of passing x which is approx $|u|$ we now only pass values $|N|$ and B which are only 1024 bit security parameters.

The above PoE works for exponentiation of the form 2^T with fixed base 2 or for powers of two. The exponentiation that we are trying to prove is of the form g^x so we must alter Wes[18] and we again turn to the work of Boneh et al []. Sparing details we find that we can use NI-PoKE* to fit our needs for plasma.

Hash to Prime

There are a few methods for assigning prime numbers and two places in Plasma vector that this will be required. In the PoKE scheme detailed above, B is the output of a H_{prime} function, and all of the indices of our sparse VC require an output from a H_{prime}. In order to appropriately choose our output spaces, we need to observe the distribution of primes in the number line. Luckily prime numbers occur quite frequently. To illustrate this we can roughly estimate the density of primes using $n/\ln(n)$. Given 10^{50} numbers we might expect $10^{50}/\ln(10^{50}) = 10^{40}$ primes.

A strategy for H_{prime} for PoKE is using an iterative function and a counter to hash from the domain Z_{2^λ}]to prime [CS99][FT14]. We can use a function that assigns smaller primes to the vector indices as was described by [] with, $f(i) = 2(i + 2) * \log_2(i + 2)^2$. If primality checking becomes too expensive then we may assign arbitrary large integers that have prime factorizations[TODO check this]. Additionally the prover could provide a nonce s.t. $H(nonce||index) = l$ with $l \in Primes(\lambda)$. This would reduce the primality checks to just one instead of λ as is with the iterative counter function method.

A security requirement of the Wesolowski PoKE is that $B > n$ where n is the length of the vector A. If $n = 2^{48}$ primes then we will have to ensure that $B > 10^{15}$

and prime. We can make a criteria of the prime selection; choose primes within the range of $[2^{48}, 2^{62}]$. This will add extra cycles to the hash to prime function as we must decline valid primes that are too small. [Todo probability stats on increased complexity to htp].

Bitmaps

Bitmaps are a way to compress a large set of integers into an array of $\{0, 1\}$. [Todo]

Plasma Vector Spec

Below we outline how vector based accumulators may be implemented in Plasma Cash.

Deposit

This is done in the same way as Plasma Cash with an onchain deposit to the plasma parent contract. The only difference here will be the assignment of a unique vector that will identify a new range of tokens. We have 2^{48} primes to represent $=$ 256-bit values (keccak hash) in a vector of size 2^{40}.

Send

The send format allows for an owner of a batch of coins to be able to generate a transaction that assigns all or a subset of their owned coins to a new owner.

A note on plasma cash history: If coins by index [1,2,3,4] are sent in one tx, and say they all have individual transaction histories, you still need to send the individual histories to the receiver before the group tx, then if [2,3] are sent on again by the receiver, the next recipient needs the individual history for that [1,2,3,4] tx and the current [2,3] tx.

Further example.

b0 = tx1 send 1 A -¿ B...tx2 send 2 A -¿ B...tx3 send 3 A -¿ B...tx4 send 4 A -¿ B b1 = tx5 send 1234 B -¿ C b2 = tx6 send 34 C -¿ D

TX history for B = tx[1,2,3,4] TX history for C = tx[1,2,3,4,5] TX history for D = tx[1,2,3,4,5,6]

In plasma vector we are using an accumulator that stores the current state in each slot at all times rather than null values in the sparse merkle tree leaves. This means that only the latest valid state transition of an owned coin (namely, the transition that sent to coin to the owner) is needed. This will further be discussed in the Coin Exit Challenge section below.

Exits

When using a sparse merkle tree to store transaction data in the accumulator, you must store null values for every block that nothing transacts for a given index. This would require that exits present the inclusion proof of their receive transaction against a previous block. Since our vector accumulator always holds the latest state

of an index (the owners receive tx) then exits should be initiated against the current block commitment in the parent chain. The only exception to this rule will be for challenges detailed below.

<div align="center">*Batched Exits*</div>

A challenge of Plasma Vector is the ability to mass exit a large batch of coins. If we have 2^{40} vector components, and we want to represent a Finney (0.001 ether), we need 1000 ids per ether. So we can represent $2^40/1000 = 1,099,511,627.776 ether. There are a total of 103,893$ = 1,099,511.62 ether. That's about 1 percent of the current total supply on our chain with a granularity of being able to send as low as 0.000001 ether. This does not mean that the exit cost allows for such a small denomination. We must still gather exact gas costs for optimal and challenge based exits to determine what the lowest denomination of the plasma chain may securely be.

[todo: illustrate what happens if you have denominations worth less than the cost to exit]

SSTORE requires 20k gas per 32 byte word, so if we cram 4 * 64 bit numbers in thats 5k*n. If we have a granularity of 0.00001 ether it would require 10,000 IDs to exit 1 ether worth of coins. If we need to register just 10k IDs in one exit then this would require 50mm gas and would not be practical. We must allow for individual ids to be batched per exit. We mark these batches by owned ranges. Now we may have to exit a total of the number of fragmented owned ranges of ids that a single user has. Atomic swaps may reduce the fragments but without further research it is hard to say if there is a proper strategy to defragment.

When an owners client begins to fragment too much we could suggest an option to exit all ranges and re-enter with less fragmented denominations.

<div align="center">*Challenges*</div>

<div align="center">*Coin Exit challenges*</div>

When an adversary attempts to exit with an honest users id, they may present an inclusion proof against an older accumulator commitment to reveal any future commitments to invalid state transitions.

challengSpentCoin

challege...

Operator challenges challengeInvalidHashToPrime challenge

<div align="center">**Smart Contract**</div>

deposit

submitBlock

[TODO]

<div align="center">**References**</div>

1

2

3

4

5