Plasma Vector

Nathan Ginnever

December 16, 2018

Introduction

This work presents an application of cryptographic universal accumulators [5] that was expanded upon in the blockchain setting by Bunnz 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 the list and we explore why this is a desired property of accumulators in the blockchain application setting. Using VC with message spaces we show how to construct a key-value map in an accumulator and its application to Cash [] history proofs. Our contribution is applying VCs to reducing the inclusion and exclusion proof size of Plasma Cash/flow [] 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.

As with previous Plasma designs, this paper will detail the *deposit*, *send*, and *exit* components. The Plasma Vector parent contract does not verify the correctness of state transitions, it only provides an immutable ledger to record accumulator commitments. We fall back on the Plasma Cash/flow 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) works like an 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. VC allow for an M-bit $\{0,1\}^\lambda$ message to be committed to each of the accumulator vector components. 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 $O(\log n)$ to open a value at a position, VC have 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 $ginG$ 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 random 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 *defragment* [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 during setup.

Update

We define now how to update our VC. Our current example assume that the message space of $M = \{0, 1\}$ for simplicity, but we can 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 != 0. We can define $A = g^{init}$ and generate and 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 non-inclusion proof however. Note that when a VC only contains 1-bit messages it takes on the form of dynamic 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 increase the amount of data that our vector components can hold to that of 160 bits for an ethereum public key. This is a pain point as a public key may have many 0 bits, and we cannot batch exclusion proofs. We continue our example to illustrate this.

Proof of Exponent Knowledge

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 Wesoloski 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. We will need the PoKCR (proof of knowledge cofactor roots) to compress our proof π size below the $|M|$ or the length of our vector.

Deposit

This is done in the same way as cashflow with an onchain token 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.

Send

The send format is described above and allows for an owner of a range of coins to be able to generate a transaction that assigns all or part of their range to a new owner.

Exit

Since we have no merge protocol, we must exit all fragmented ranges separately. Atomic swaps may help to defragment ranges. [todo expand]

Smart Contract

[TODO]

References

1
2

3
4
5