Bilinear vector commitments for stateless blockchains

Steve Thakur

Axoni Research Group

Abstract

This is meant to be a rough design for a permissioned blockchain that is *stateless* in the sense that the concerns of state storage and consensus are decoupled. In order to maintain a high throughput and a constant amount of storage for the validators, we propose a design in which the data is committed using a bilinear Vector Commitment that uses a multi-party computation performed by the validators. We will build upon our recent work on bilinear Vector Commitments ([T19]) which, in turn, is heavily influenced by the techniques of [BBF19], [Ngu05] and [Wes18].

1 Introduction

One of the primary issues that make it all but impossible to scale blockchains at the moment is that of state bloat. This problem is substantially worse for accounts-based models such as Ethereum than for UTXO-based models such as Bitcoin. In response to this, the Ethereum foundation recently announced a hard fork to a stateless model. In the preliminary version of ETH 1.x the clients will be stateless while the miners will continue to hold the full state. Their end goal is the Version 2.0 where the miners as well as clients will be stateless. Unfortunately, the road to ETH 2.0 is paved with quite a few unsolved problems. In particular, it is widely acknowledged that a stateless model would require membership proofs to be sent over the network in addition to the data and hence, with a Vector Commitment such as a Merkle tree or a Merkle Patricia trie, the bandwidth would be a bottleneck since the proofs grow logarithmically in size and more importantly, cannot be batched/aggregated. With this in mind, our goal is to design a stateless *permissioned* blockchain that hinges on a Vector Commitment with constant-sized membership proofs which can be batched or aggregated.

In our recent work ([T19]), we provided protocols to batch and aggregate multiple non-membership proofs into a single proof of constant size with bilinear accumulators. We subsequently used the accumulator to construct a sparse bilinear Vector Commitment (VC) with constant sized openings and a linear public parameter. This VC yields a key-value commitment which we intend to use to store the state of the blockchain in this design.

We also provided ways to speed up the verification of membership and non-membership proofs and to shift most of the computational burden from the Verifier to the Prover. Furthermore, we designed the protocols so that the Verifier needs a constant amount of storage for the verification of any relevant computation. Since all of the protocols are public coin, they can be made non-interactive with a Fiat-Shamir heuristic. This is particular useful in cases where there are multiple Verifiers.

We propose a design for a stateless blockchain where data is committed via bilinear Vector Commitments and multi-party computations by the validators. Following the ideas of [BBF19], we introduce an untrusted oracle node that has verifiable storage of the state of the blockchain and provides updated proofs of membership or non-membership to syncing nodes. By introducing

this new design, we aim to significantly reduce the storage requirements for nodes on the network, as well as increase performance through highly parallelizable consensus and state operations. We would also benefit from a key-value commitment that is simpler in nature than that yielded by the MPT. To this end, we intend to use the key-value commitment arising from bilinear VCs.

1.1 Types of nodes on the network

- A **client** is a person who interacts with the blockchain network and proposes transactions to add to state.
- An **oracle** is a node that has the capability to hold the latest and historical state of the blockchain. When data is requested from the oracle node, it returns the data if it exists as well as a proof to ensure data integrity. An oracle node also subscribes to updates to accumulated state. When a new state is committed to the blockchain, the accumulated state digest and s-powers are broadcast to subscribers such as oracle nodes.
- An **API** is a stateless service that makes requests to nodes in the blockchain network on behalf of the client. In order to make requests to the API, a client would authenticate themselves with user credentials. The API would then use a client's user credentials to determine if they are authorized to perform a given operation.
- A **proposer** is a node that has the capability to hold the latest state, update membership proofs for data relevant to the clients, and submit transactions on behalf of the clients to validators on the blockchain network. The proposer is an untrusted entity. As a result, transactions submitted by the proposer to validators must include a witness and requests submitted by the client to the proposer return a proof that the request was correctly and successfully submitted to the network. On a permissioned network, it would be more efficient for the proposer to aggregate the witnesses before he sends them to the validators. The proposer could also aggregate the signatures of the clients and send the aggregated signature to the validators for faster verification. It appears that BLS signtures are more efficient than ECDSA signatures for this purpose.
- A **processing validator** is a node that has the capability to participate in consensus and accumulate the global state of the network. In the intermediate model, a processing validator will hold the smart contract code in order to verify that a transaction submitted by a proposer yields a correct, agreed-upon output. While the validator will not be truly stateless, this would still mean a massive drop in his storage.

  One of our primary goals is to eventually move to a model where the processing validators do not need to hold the smart contract code either.
- A **verifying validator** is a node that has the capability to verify that the new accumulated global state of the network is correct. In addition, a verifying validator may retain historical accumulate state digests and s-powers of accumulated digests until it is confirmed that the data has been successfully stored by oracle nodes.

In theory, any node could be a verifying validator since the storage requirements are rather low (four points on the elliptic curve) as are the computational requirements (two pairings). But as always, the devil $\in$ the details.

Design Goals: We would like the storage of the validators to be constant-sized. We would also like the users to store as little as possible in addition to the data they care about and a commitment to the state, i.e. constant-sized batched membership proofs for the data elements they care about. In particular, if $\mathcal{D}_0$ is the data a user cares about, his storage should be $\mathbf{O}(\#\mathcal{D}_0)$

with a reasonably small constant.

We would like the amount of necessary interaction between the users and the validators to be at a bare minimum, thus allowing the validators to focus almost exclusively on consensus and updating the accumulated digest. This mimics the design proposed in [BBF19], although their underlying accumulator/VC is one based on groups of unkown order and is better suited for public blockchains with a lower throughput requirement.

Our construction does not eliminate the need for (server) nodes that store the entire blockchain. But it would be desirable for these nodes to lie outside the critical consensus path, i.e. it is preferable for them not to be validators. It would also be desirable to not have to trust these nodes for integrity but only for data availability. To this end, we sketch a proposition for a stateless blockchain using bilinear accumulators/VCs.

1.2 Notations and terminology

As usual, $\mathbb{F}_q$ denotes the finite field with q elements for a prime power q and $\overline{\mathbb{F}}_q$ denotes its algebraic closure. $\mathbb{F}_q^*$ denotes the cyclic multiplicative group of the non-zero elements of $\mathbb{F}_q$. For univariate polynomials $f(X), g(X) \in \mathbb{F}_q[X]$, we denote by $\gcd(f(X), g(X))$ the unique *monic* polynomial $h(X) \in \mathbb{F}_q[X]$ that generates the principal ideal of $\mathbb{F}_q[X]$ generated by $f(X)$ and $g(X)$.

Definition 1.1. Batching and aggregation: Following the terminology of [BBF19], we use the term *batching* for the action of creating a single membership (or non-membership) witness for multiple data elements. *Aggregation* refers to the action of creating a single membership or non-membership proof for the data elements using individual witnesses that have already been created.

Neither of these mechanisms is afforded by Merkle trees, which is a primary reason for exploring other families of cryptographic accumulators and Vector Commitments.

Definition 1.2. An argument system between a Prover and a Verifier is *non-interactive* if it consists of a single round.

Definition 1.3. An argument of knowledge is said to be *public coin* if all challenges sent from the Verifier to the Prover are chosen uniformly at random and independently of the Prover's messages.

If the challenges are public coin, any interactive argument of knowledge can be converted into a non-interactive argument using a Fiat-Shamir heuristic ([FS86]). We now briefly introduce pairings.

Definition 1.4. For abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, a *pairing*

$$\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$$

is a map with the following properties.

1. Bilinearity: $\mathbf{e}(x_1 + x_2, y_1 + y_2) = \mathbf{e}(x_1, y_1) * \mathbf{e}(x_1, y_2) * \mathbf{e}(x_2, y_1) * \mathbf{e}(x_2, y_2)$
$\forall\ x_1, x_2 \in \mathbb{G}_1,\ y_1, y_2 \in \mathbb{G}_2$.
2. Non-degeneracy: The image of $\mathbf{e}$ is non-trivial.
3. Efficient computability.

In pairing-based cryptography, we typically work in settings where the groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ are cyclic of order p for some 256-bit prime p so as to have a 128-bit security level. Such pairings $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$ are classified into three types:

- Type I: $\mathbb{G}_1 = \mathbb{G}_2$.
- Type II: $\mathbb{G}_1 \neq \mathbb{G}_2$ but there is an *efficiently computable* isomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$.
- Type III: There is no *efficiently computable* isomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$.

In practice, the groups $\mathbb{G}_1, \mathbb{G}_2$ are cyclic subgroups of the p-torsion subgroup of some pairing-friendly elliptic curve over a prime field $\mathbb{F}_l$ and $\mathbb{G}_T$ is the group of p-th roots of unity in the algebraic closure $\overline{\mathbb{F}_l}$. The pairing is usually the (alternating) Weil pairing which is efficiently computable with Miller's algorithm. If the elliptic curve is supersingular, a symmetric pairing on the group of p-torsion points can be derived by composing the Weil pairing with an appropriate endomorphism of the elliptic curve. No such symmetric pairing exists for ordinary elliptic curves (as far as we know).

Accumulation of data: The accumulation is performed using the *distributed* private key. This hinges on at least one of the validators being honest enough not to collude with the other validators to retrieve the private key. We describe the key generation here.

Let N be the total number of processing validators and t the quorum size, i.e. the minimum number of validators necessary to process data. Let $V_1, \cdots, V_N$ be the processing validators. Each validator V_i chooses a random element $s_i \in \mathbb{F}_p^*$. The product

$$s = \prod_{i=1}^{N} s_i$$

will function as the private key. Whenever a new datum d is to be accumulated, the state changes from A to A^{s+d}. Each validator V_i changes the input by raising it to the s_i-th power and the last validator to do this multiplies

$$A^s = A^{\prod_{i=1}^{N} s_i}$$

by A^d. For validator V_i to see the piece s_j $(j \neq i)$, validator V_i would need to solve the discrete logarithm problem (DLP) in the group $\mathbb{G}_1$. So this is secure under the assumption of hardness of the DLP.

1.3 Cryptographic assumptions

We state the computationally infeasible problems that the security of our constructions depends on.

Definition 1.5. ***n*-strong Diffie Hellman assumption:** *Let $\mathbb{G}$ be a cyclic group of prime order p generated by an element g, and let $s \in \mathbb{F}_p^*$. Any probabilistic polynomial-time algorithm that is given the set $\{g^{s^i} : 1 \leq i \leq n\}$ can find a pair $(a, g^{1/(s+a)}) \in \mathbb{F}_p^* \times \mathbb{G}$ with probability at most $\mathbf{O}(\frac{1}{p})$.*

Definition 1.6. Knowledge of exponent assumption:. *Let $\mathbb{G}$ be a cyclic group of prime order p generated by an element g, and let $s \in \mathbb{F}_p^*$. Suppose there exists a PPT algorithm $\mathcal{A}_1$ that given the set $\{g^{s^i}, g^{s^i \alpha} : 1 \leq i \leq n\}$, outputs a pair $(c_1, c_2) \in \mathbb{G} \times \mathbb{G}$ such that $c_2 = c_1^{\alpha}$. Then there exists a PPT algorithm $\mathcal{A}_2$ that, with overwhelming probability, outputs a polynomial $f(X) \in \mathbb{F}_p[X]$ of degree $\leq n$ such that $c_1 = g^{f(s)}, c_2 = g^{\alpha f(s)}$.*

Several cryptosystems including the BLS signature scheme and Groth's zk-SNARKs hinge on these assumptions. So our cryptographic assumptions are not too strong.

1.4 Mechanism design assumptions

Our construction hinges on the assumption that we will never have a quorum of dishonest validators. So, if the total number of processing validators is N and the quorum size is t, we must assume that the number of dishonest processing validators is $\leq t-1$. In other words, at least $N-t+1$ validators must be honest not to collude with other validators to retrieve the secret key s.

This assumption can be weakened to a degree depending on the use case. But in the general setting where all of the validators are just as likely to be malicious, we need this assumption. This is the primary reason why this construction would be a non-starter for an open blockchain. On the other hand, the throughput requirements for a public blockchain are generally far lower than what we face. Hence, open stateless blockchains might more feasibly build on RSA or class group Vector Commitments.

1.5 Shamir's secret key-sharing protocol:

One of the issues we face is that a processing validator node could go offline at some point. Hence, there needs to be an understanding in advance that in the absence of any validator, the remaining validators can and will retrieve his piece of the private key in order to process future transactions. To this end, we use Shamir's secret key-sharing scheme for each piece s_i $(i = 1, \cdots, N)$. Let $V_1, \cdots, V_N$ be the processing validators, t the quorum-size and $s_i \in \mathbb{F}_p^*$ the part of the secret key held by V_i.

The validator V_i chooses an t-degree polynomial

$$f_i(X) = s_i + \sum_{j=1}^{N} c_{i,j} X^j \in \mathbb{F}_p[X].$$

He then computes and sends $f_i(j) \in \mathbb{F}_p$ to the validator V_j $(1 \leq j \leq N,\ j \neq i)$. Thus, any collection of t parties can collaborate to retrieve s_i but no fewer than t parties can collude to do so except with negligible probability.

In order for this to work, each validator V_j must store the set $f_i(j)$ $(1 \leq i \leq N,\ j \neq i)$ in addition to his own piece of the secret key. All of this computation and sharing occurs before the genesis block. Shamir's scheme is known to be information theoretically secure. But our construction hinges on the assumption that at least one of the parties left at any given time is honest enough not to collude with the other parties to derive the secret key.

Suppose a validator V_j disappears and a new node V_{N+1} is elevated to the status of a processing validator. The remaining validators on the network agree on a single element $c \in \mathbb{F}_p^*$ and send the elements $f_i(j) * c^{N-1} \in \mathbb{F}_p^*$ to V_{N+1}. The latter then derives the element $s_{N+1} := s_i * c^{N-1} \in \mathbb{F}_p^*$ and uses it as his portion of the secret key. The validators V_j $(1 \leq j \leq N, j \neq i)$ then change their pieces of the secret keys from s_j to $c^{-1}s_j$. The secret key remains unchanged.

To deal with a scenario where a validator node *temporarily* disappears, it might be better for each node V_j to share not just s_j but the entire set

$$\{s_j, \cdots, s_j^n\}$$

where n is an upper bound on the block size, using the Shamir key sharing scheme. For security, it is necessary for V_j to choose n completely random polynomials in $\mathbb{F}_p[X]$. We will circle back to this in section 2.

1.6 Membership proofs:

We briefly review how membership witnesses work in bilinear accumulators. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of order p for some prime p such that there exists a pairing $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$ which is *bilinear*, *non-degenerate* and *efficiently computable*. Fix generators g_1, g_2 of the cyclic groups $\mathbb{G}_1, \mathbb{G}_2$ respectively. Then $\mathbf{e}(g_1, g_2)$ is a generator of $\mathbb{G}_T$.

In practice, the groups $\mathbb{G}_1, \mathbb{G}_2$ are cyclic subgroups of the p-torsion subgroup of some pairing-friendly elliptic curve over a prime field $\mathbb{F}_l$ of bit-size 256 and $\mathbb{G}_T$ is the group of p-th roots of unity in the algebraic closure $\overline{\mathbb{F}_l}$. The pairing is usually the (alternating) Weil pairing which is efficiently computable with Miller's algorithm. It appears that the pairings that are best suited for our purposes are those of type III, i.e. those for which there is no efficiently computable isomorphism $\mathbb{G}_1 \longrightarrow \mathbb{G}_2$.

For a data set $\mathcal{D} = \{d_1, \cdots, d_n\}$, we define the accumulated digest

$$\mathrm{Acc}(\mathcal{D}) := g_1^{\prod_{d \in \mathcal{D}} (d+s)} \in \mathbb{G}_1.$$

For a subset $\mathcal{D}_0 \subseteq \mathcal{D}$, the witness for $\mathcal{D}_0$ is defined by

$$\mathrm{Wit}(\mathcal{D}_0) := g_1^{\prod_{d \in \mathcal{D} \setminus \mathcal{D}_0} (d+s)}.$$

The Verifier then needs to check whether

$$\mathrm{Wit}(\mathcal{D}_0)^{\prod_{d_0 \in \mathcal{D}_0} (d_0+s)} = \mathrm{Acc}(\mathcal{D}).$$

Because of the bilinearity of the pairing, it is equivalent and usually more efficient to test the following equation:

$$\mathbf{e}(\mathrm{Wit}(\mathcal{D}_0), g_2^{\prod_{d_0 \in \mathcal{D}_0} (d_0+s)}) = \mathbf{e}(\mathrm{Acc}(\mathcal{D}), g_2).$$

The exponent $\prod_{d \in \mathcal{D}} (s + d)$ can be interpreted as a degree $\#\mathcal{D}$ polynomial in the variable s. The coefficients of the polynomial are computed with a run time of $\mathbf{O}(n \log(n))$ using the Fast Fourier transform. Furthermore, the set $\mathbb{F}_p[X]$ of polynomials with $\mathbb{F}_p$-coefficients is a principal ideal domain whose maximal ideals are those generated by the irreducible polynomials. For a data set $\mathcal{D}$, the polynomial $f(X) = \prod_{d \in \mathcal{D}} (X + d)$ is monic of degree $n = |\mathcal{D}|$. Let c_i denote the coefficient of X^i, i.e. $f(X) = \sum_{i=0}^{n} c_i X^i$. The coefficients can be computed in run time $\mathbf{O}(n \log(n))$ using the Fast Fourier transform. The elements

$$g_1^{f(s)} = \prod_{i=0}^{n} (g_1^{s^i})^{c_i}, \quad g_2^{f(s)} = \prod_{i=0}^{n} (g_2^{s^i})^{c_i}$$

can then be computed by any party that possesses the elements $\{g_1^{s^i}, g_2^{s^i} : 0 \leq i \leq n\}$. The security of the bilinear accumulator hinges on the n-Strong Diffie Hellman assumption.

The straightforward approach would be for the Verifier to compute $g_2^{\prod_{d_0 \in \mathcal{D}_0} (d_0+s)} \in \mathbb{G}_2$ in order to verify the equation

$$\mathbf{e}(\mathrm{Wit}(\mathcal{D}_0), g_2^{\prod_{d_0 \in \mathcal{D}_0} (d_0+s)}) = \mathbf{e}(\mathrm{Acc}(\mathcal{D}), g_2).$$

However, this involves computing the coefficients c_i $(0 \leq i \leq |\mathcal{D}_0|)$ of the polynomial $\prod_{d_0 \in \mathcal{D}_0} (d_0 + s)$. The fastest known algorithm (the Fast Fourier transform) for this has run time complexity $\mathbf{O}(|\mathcal{D}_0| \log(|\mathcal{D}_0|))$ - followed by the exponentiations $(g_2^{s^i})^{c_i}$ that have run times $\mathbf{O}(\log(c_i))$. Furthermore, this makes it necessary for the Verifier to possess the entire set $\{g_1, g_1^s, \cdots, g_1^{s^n}\}$ which is just unthinkable. To address this, we provide the protocols `PoE` and `PoKE` for bilinear accumulators ([T19]) which achieve three goals.

1. They speed up the verification process by replacing some exponentiation operations by polynomial division in $\mathbb{F}_p[X]$ which is substantially cheaper.

2. Secondly, they shift most of the computational burden from the Verifier to the Prover. This is useful in settings where the Prover has more computational power at his disposal.

3. Finally, they reduce the initial information necessary to be broadcasted to the Verifier to the set $\{g_1, g_1^s, g_2, g_2^s\}$ of size four. This allows the Verifier to have a constant sized storage to verify any relevant computation.

Aggregation of membership witnesses with the distributed private key: We describe the process of aggregating two witnesses. This generalizes to arbitrarily many witnesses via induction. Let $\mathcal{D}_1$, $\mathcal{D}_2$ be two data sets and w_1, w_2 the corresponding membership witnesses with respect to the accumulated digest A. Set

$$f_i(X) := \prod_{d \in \mathcal{D}_i} (X + d), \quad i = 1, 2.$$

The processing validators first compute polynomials $e_1(X), e_2(X) \in \mathbb{F}_p[X]$ such that

$$e_1(X) f_1(X) + e_2(X) f_2(X) = 1, \quad \deg e_i(X) < \deg f_{3-i}(X).$$

They then compute $w_1^{e_2(s)}, w_2^{e_1(s)}$ using the distributed private key and set $w_{1,2} := w_1^{e_2(s)} w_2^{e_1(s)}$. By construction,

$$w_{1,2}^{f_1(s) f_2(s)} = A$$

and hence, $w_{1,2}$ is a membership witness for the union $\mathcal{D}_1 \cup \mathcal{D}_2$.

1.7 Protocols PoE and PoKE

In order to make this document more self-contained, we state the protocols `PoE` and `PoKE` for bilinear accumulators from [T19]. We refer the reader to [T19] for proofs of security and further details.

Protocol 1.1. *Protocol* `PoE` *(proof of exponent) for pairings (interactive version):*

Parameters : A pairing $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$ of groups of prime order p; generators g_1, g_2 of $\mathbb{G}_1$, $\mathbb{G}_2$ respectively; a secret element $s \in \mathbb{F}_p^*$
Inputs: $a, b \in \mathbb{G}_1$; a polynomial $f(X) \in \mathbb{F}_p[X]$ of degree $\leq n$
Claim: $a^{f(s)} = b$

1. The Verifier sends an element α (the challenge) to the Prover.

2. The Prover computes a $h(X)$ and a element β such that $f(X) = (X + \alpha) h(X) + \beta$. The Prover sends $Q := a^{h(s)}$ to the Verifier.

3. The Verifier computes $\beta := f(X) \pmod{(X+\alpha)}$ and accepts if and only if the equation

$$\mathbf{e}(Q, g_2^{s+\alpha}) * \mathbf{e}(a, g_2^{\beta}) = \mathbf{e}(b, g_2)$$

holds. □

Protocol 1.2. *Protocol for the membership of a set.*

Parameters : A pairing $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$ of groups of prime order p; generators g_1, g_2 of $\mathbb{G}_1$, $\mathbb{G}_2$ respectively; a secret element $s \in \mathbb{F}_p^*$
Inputs: The accumulated set $\mathcal{D}$; a data set $\mathcal{D}_0$.
Claim: $\mathcal{D}_0 \subseteq \mathcal{D}$.

1. The Prover computes the polynomial $f_0(X) := \prod_{d_0 \in \mathcal{D}_0} (X + d_0)$.

2. The Prover computes

$$\mathrm{Wit}(\mathcal{D}_0) := g_1^{\prod_{d \in \mathcal{D} \setminus \mathcal{D}_0} (d+s)}.$$

3. The Prover sends the Verifier a non-interactive `PoE` for $\mathrm{Wit}(\mathcal{D}_0)^{f_0(s)} = \mathrm{Acc}(\mathcal{D})$.
4. The Verifier computes $f_0(X)$ and accepts if and only if the `PoE` withstands this scrutiny. □

The protocol `PoE` allows us to pass the burden of computation from the Verifier to the Prover (who would usually be doing those computations anyway). In particular, one validator could perform the computations and send a succinct `PoE` to the other validators on the network.

Protocol 1.3. *Protocol* `PoKE` *(proof of knowledge of the exponent) for bilinear accumulators (interactive version):*

Parameters: A pairing $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$; Inputs: Elements $a, b \in \mathbb{G}_1$
Claim: The Prover possesses a polynomial $f(X) \in \mathbb{F}_p[X]$ such that $a^{f(s)} = b$

1. The Verifier sends a challenge $\alpha \in \mathbb{F}_p^*$ to the Prover.
2. The Prover computes the polynomial $h(X) \in \mathbb{F}_p[X]$ and the element $\beta \in \mathbb{F}_p$ such that $f(X) = (X+\alpha)h(X) + \beta$. The Prover then computes $Q := a^{h(s)}$ and sends Q, β to the Verifier.
3. The Prover computes $\widetilde{g}_2 := g_2^{f(s)}$ and sends it to the Verifier.
4. The Verifier then verifies the two equations

$$\mathbf{e}(a, \widetilde{g}_2) = \mathbf{e}(b, g_2); \quad \mathbf{e}(Q, g_2^{s+\alpha}) * \mathbf{e}(a^{\beta}, g_2) = \mathbf{e}(b, g_2).$$

He then accepts if and only if both equations hold. □

Protocol 1.4. *Protocol* `PoKE` *(proof of knowledge of the exponent) for type* III *bilinear accumulators:*

Parameters: A type III pairing $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$; Inputs: Elements $a, b \in \mathbb{G}_1$
Claim: The Prover possesses a polynomial $f(X) \in \mathbb{F}_p[X]$ such that $a^{f(s)} = b$

1. The Prover computes $Q := g_2^{f(s)}$ and sends it to the Verifier.
2. The Verifier accepts if and only if $\mathbf{e}(a, Q) = \mathbf{e}(b, g_2)$. □

2 Accumulatation of data

Processing additions with the distributed private key: Let $V_1, \cdots, V_N$ be the processing validators and let s_i denote the piece of the private key that V_i possesses. The private key s is given by $s := \prod_{i=1}^{N} s_i$.

Let A be the accumulated digest at the end of a certain block. Let A_1 be the accumulated digest after a data set $\mathcal{D}_{\text{del}}$ has been deleted. So, we have

$$A = A_1^{\prod_{d \in \mathcal{D}_{\text{del}}} (s+d)}.$$

Let $\mathcal{D}_{\text{add}}$ be the data set to be added in this block. Write $\#\mathcal{D}_{\text{add}} = N * q + r, \;\; r \leq N-1$. For simplicity, assume $r = 0$ for now. We proceed as follows.

Step 1. The Validator V_i $(i = 1, \cdots, N)$ generates the set

$$\{A_1^{s_i^{q(i-1)+1}}, \cdots, A_1^{s_i^{qi}}\}$$

and sends it to $V_{i+1 \pmod N}$

Step 2. The Validator V_i $(i = 1, \cdots, N)$ generates the set

$$\{A_1^{(s_{i-1}s_i)^{q(i-2)+1}}, \cdots, A_1^{(s_{i-1}s_i)^{q(i-1)}}\}$$

and sends it to $V_{i+1 \pmod N}$.

...

...

...

Step N. The Validator V_i $(i = 1, \cdots, N)$ generates the set

$$\{A_1^{s^{qi+1}}, \cdots, A_1^{s^{q(i+1)}}\}$$

and sends it to $V_{i+1 \pmod N}$.

Proceeding in this manner, in N Steps, the validators have computed the set

$$\{A_1, A_1^s, \cdots, A_1^{s^{\#\mathcal{D}_{\text{add}}}}\}$$

which they will need to compute the new block header. This set (of size $257 * \#\mathcal{D}_{\text{add}}$-bits) is stored by the storage server node so that the validators can maintain a constant amount of storage. These computations are highly parallelizable. The time needed for all of these computations has an upper bound of

$$\frac{256 * (\text{block size}) * (\text{time consumed by a point addition/doubling in } \mathbb{G}_1)}{\#\text{ parallel processors that each validator possesses}}$$

while the time consumed by the transmission of information is

$N * (\text{time needed for one validator to send a set of size } 64 * q \text{ bytes to another validator})$.

The validators then (independently) compute the polynomial

$$f(X) := \prod_{d \in \mathcal{D}_{\text{add}}} (X + d) = \sum_{i=0}^{\#\mathcal{D}_{\text{add}}} c_i X^i \in \mathbb{F}_p[X]$$

using the Fast Fourier transform. They also derive a challenge $\alpha \in \mathbb{F}_p^*$ using a Fiat-Shamir heuristic and compute $h(X) \in \mathbb{F}_p[X],\ \beta \in \mathbb{F}_p^*$ such that

$$f(X) = (X + \alpha)h(X) + \beta.$$

Now, any validator (such as an elected leader) that sees the broadcasted sets

$$\{A_1, A_1^s, \cdots, A_1^{s^{\#\mathcal{D}_{\text{add}}}}\},\ \ \mathcal{D}_{\text{add}}$$

can *autonomously* compute the new accumulated digest

$$\mathbf{new}(A) := A_1^{f(s)} = \prod_{i=0}^{\#\mathcal{D}_{\text{add}}} (A_1^{s^i})^{c_i}$$

and the element $Q := A_1^{h(s)}$ of $\mathbb{G}_1$. To verify that the computations were performed accurately, it suffices for the verifying validators and other nodes on the network to check whether

$$Q^{s+\alpha} * A_1^{\beta} = \mathbf{new}(A).$$

A more skeptical node could verify the equation

$$\mathbf{e}(\mathbf{new}(A), g_2) = \prod_{i=0}^{N} \mathbf{e}(A_1^{c_i}, g_2^{s^i})$$

instead. The accumulated digest for the new block is then updated to $\mathbf{new}(A)$.

One of the validators (such as an elected leader) computes the element $\widetilde{g}_2 := g_2^{f(s)} \in \mathbb{G}_2$. This will serve as a non-interactive `PoK` of the polynomial $f(X)$. Any node that needs to verify that $\mathbf{new}(A)$ was computed by a series of insertions (as opposed to deletions) from A_1 simply needs to verify the equation

$$\mathbf{e}(A_1, \widetilde{g}_2) = \mathbf{e}(\mathbf{new}(A), g_2).$$

We expect there to be a some network overhead arising fromn the Validators communicating the sets of size $64 * q$ bytes to each other. However, these sets are completely independent of the data $\mathcal{D}_{\text{add}}$ to be inserted. The sets

$$\{A_1, A_1^s, \cdots, A_1^{s^{\#\mathcal{D}_{\text{add}}}}\},\ \ \mathcal{D}_{\text{add}}$$

are eventually dumped onto the oracle nodes. Consequently, the storage of the validators remains constant. Furthermore, a user node that needs these sets to update its proofs would have to query this oracle node instead of the validators. The (untrusted) oracle node could required to provide verifiable proofs of storage to the validators at periodic intervals.

Furthermore, we are still in th process of acquiring concrete numbers for the network latency which depends heavily on the use case. In the event that the network latency is high, the validators could compute a larger set

$$\{A, A^s, \cdots, A^{s^N}\}$$

of s^i-powers of the latest accumulated digest A. This would still allow them (or any quorum of processing validators) to compute the accumulated sigest resulting from inserting N or fewer data elements.

Deletions: Let $\mathcal{D}_1, \cdots, \mathcal{D}_r$ be disjoint data sets from different users that need to be deleted and let $w_1, \cdots, w_r$ be the corresponding witnesses that the users send to the validators. Write $f_i(X) := \prod_{d \in \mathcal{D}_i} (X + d)$ and $\mathcal{D}_{\text{del}} := \bigcup_{i=1}^{r} \mathcal{D}_i$. Then

$$w_i^{f_i(s)} = A, \;\; i = 1, \cdots, r$$

and

$$w^{\prod_{i=1}^{r} f_i(s)} = A$$

where w is the membership witness of $\mathcal{D}_{\text{del}}$. The processing validators $V_1, \cdots, V_N$ compute the sets

$$\{w_i, w_i^s, \cdots, w_i^{s^{\#\mathcal{D}_i}}\}, \;\; i = 1, \cdots, r.$$

They then compute polynomials

$$e(X), e_1(X), \cdots, e_r(X) \in \mathbb{F}_p[X], \;\; \deg e_i(X) < \deg f_i(X) = \#\mathcal{D}_i$$

such that

$$w = A^{e(s)} \prod_{i=1}^{r} w_i^{e_i(s)}.$$

Now w may be easily computed using the sets

$$\{w_i, w_i^s, \cdots, w_i^{s^{\#\mathcal{D}_i}}\}, \;\; i = 1, \cdots, r$$

and

$$\{A, A^s, \cdots, A^{s^{\#\mathcal{D}_{\text{del}}}}\}.$$

The element w is the new accumulated digest after the data set $\mathcal{D}_{\text{del}}$ has been deleted.

One of the validators (such as an elected leader) computes the element $\widetilde{g}_2 := g_2^{\prod_{i=1}^{r} f_i(s)} \in \mathbb{G}_2$. This will serve as a non-interactive PoK of the polynomial $f(X)$. Any node that needs to verify that $\mathbf{new}(A)$ was computed by a series of deletions (as opposed to insertions) from A simply needs to verify the equation

$$\mathbf{e}(A, g_2) = \mathbf{e}(\mathbf{new}(A), \widetilde{g}_2).$$

Trade-off between speed and bandwidth: Alternatively, instead of the validators computing the sets

$$\{w_i, w_i^s, \cdots, w_i^{s^{\#\mathcal{D}_i}}\}, \;\; i = 1, \cdots, r,$$

the users (who possess these sets) could send them these sets over the network. This would naturally speed up the computation. Furthermore, the communication needed *between the validators* would reduce to that required for computing the set

$$\{A, A^s, \cdots, A^{s^{\#\mathcal{D}_{\text{del}}}}\}$$

with the distributed private key, thus making the communication complexity *between the processing validators* for deletions similar to that needed for insertions. However, it would increase the communication complexity *between the users and the validators*. So it is a trade-off.

2.1 Dealing with the disappearance of a validator node

One of the issues we face (in practice) is that a validator node V_i might go offline at any point. We briefly discuss how this scenario can be dealt with.

Case 1. If the validator V_i disappears *forever* or seems *utterly unreliable*, the other validators could look up the shares $f_i(j)$ of the key s_i that V_i distributed to them before the genesis block. They send these shares $f_i(j)$ to a suitable node V_{N+1} that is now elevated to the status of a processing validator. The node V_{N+1} then retrieves the key s_i and uses it to process data from that point onward. The security of the blockchain still hinges on at least one of every t validators being honest where t is the quorum size.

Case 2. If the disappearance of the node V_i is *temporary*, any t of the validators could use their shares $f_i(j)$ to perform the computations that V_j is supposed to compute to derive the set

$$\{A, A^s, \cdots, A^{s^{\text{block size}}}\}.$$

There are two ways to go about this and the choice entails a trade-off between the throughput and the storage of the processing validators.

Option (i). During the setup (before the genesis block), each validator V_j distributes not only s_j but also each power s_j^k, $1 \leq k \leq$ block size to the other validators using a Shamir t-threshold key-sharing scheme. This would speed up the computations and keep them highly parallelizable. But the storage of each validator would blow up to roughly

$$256 * (\#\text{ Validators}) * (\text{block size}) \text{ bits.}$$

For instance, with 10 validators and a block size of 2000, this amounts to 64 KB (which does not seem too bad)

Option (ii). A collection of t validators use their shares $f_i(j)$ of s_i to compute the set

$$\{A, A^{s_i}, \cdots, A^{s_i^{\text{block size}}}\}.$$

This is a substantially slower process and would mean a lower throughput. But the blockchain "still proceeds" until the node V_j comes back online. **Need some concrete numbers here but Option (i) seems better**.

We sketch the procedure for Option (i). Let V_i be a validator node that *temporily* goes offline. Let $\mathcal{Q}$ be a subset of size t of the validators, where t is the quorum size. Let n be an upper bound on the size of a block. Let A be the accumulated digest at the end of the last block.

During the setup, the validator V_j distributes the set $\{s_i, \cdots, s_i^n\}$ to the other validators using a (t, N)-threshold Shamir key sharing scheme. For each power s_i^k, he chooses a polynomial

$$f_{i,k}(X) = s_i^k + \sum_{l=1}^{t-1} c_{i,k,l} X^j \in \mathbb{F}_p[X]$$

of degree $t-1$ and sends the set

$$\{f_{i,1}(j), \cdots, f_{i,n}(j)\}$$

to the validator V_j $(1 \leq j \leq N,\ j \neq i)$. Now, by Lagrange interpolation,

$$s_i^k = f_{i,k}(0) = \sum_{j \in \mathcal{Q}} f_{i,k}(j) * e_{i,j,k}$$

for some $e_{i,j,k} \in \mathbb{F}_p$. The elements $e_{i,j,k}$ are known to all of the validators in $\mathcal{Q}$ but $f_{i,k}(j)$ is known only to the validator V_j. For brevity, we write $s_{i,j,k} = f_{i,k}(j) * e_{i,j,k}$. Now, to compute the set

$$\{A, A^{s_i}, \cdots, A^{{s_i}^n}\},$$

the validators in $\mathcal{Q}$ compute the sets

$$\{A^{s_{i,j,1}}, \cdots, A^{s_{i,j,n}}\},\ \ j \in \mathcal{Q}.$$

Now, the elements

$$A^{{s_i}^k} = \prod_{j \in \mathcal{Q}} A^{s_{i,j,k}},\ \ k = 1, \cdots, n$$

can be easily computed since they merely entail computing products of t elements of $\mathbb{G}_1$. These computations are highly parallelizable and can be performed without any of the validators in $\mathcal{Q}$ revealing their shares $f_{i,k}(j)$ of the elements s_i^k to each other.

The (effective) time taken by the validators in $\mathcal{Q}$ to compute

$$\{A, A^{s_i}, \cdots, A^{{s_i}^n}\}$$

is roughly

$$\frac{\text{(block size)} * (0.45)}{\text{\# parallel processors each validator possesses}}\ ms.$$

This does not account for the time taken by the validators transmitting their computed sets to each other.

The same process can be used if a set of processing validators (rather than a single validator) go offline *temporarily*. It puts a higher computational burden on the validators in the set $\mathcal{Q}$ and is slower than the computation would have been if all processing validators were available. But the blockchain does not grind to a halt, which is our primary concern.

Commitment: It is necessary to ensure that the validator V_j is honest when he distributes the shares of his secret to the other validators. One way to achieve this is by having the validator "commit" these shares. We describe the procedure here.

During the setup (before the genesis block), the validator V_j computes and publishes each of the elements

$$h_{i,j,k} := g_1^{f_{i,k}(j)} \in \mathbb{G}_1,\ i = 1, \cdots, N.$$

Let $\mathcal{Q}$ be any quorum of validators. By Lagrange interpolation, we have

$$s_i^k = f_{i,k}(0) = \sum_{j \in \mathcal{Q}} f_{i,k}(j) * e_{i,j,k}$$

for some elements $e_{i,j,k} \in \mathbb{F}_p$ that all validators in $\mathcal{Q}$ are aware of. For each integer k, to verify that the shares $f_{i,k}(j)$ were distributed honestly, the following suffice:

1. The validators check whether

$$g_1^{s_i^k} = \prod_{i \in \mathcal{Q}} h_{i,j,k}^{e_{i,j,k}}.$$

This is *publicly* verifiable.

2. Each validator V_j $(i = 1, \cdots, N)$ verifies that $h_{i,j,k} = g_1^{f_{i,k}(j)}$.

2.2 Updating membership witnesses

A major advantage that algebraic accumulators possess over Merkle trees is that witnesses can be batched/aggregated together. We briefly recall how membership witnesses work with bilinear accumulators. Let $\mathcal{D}_0$ be a data set and let w_0 be its witness. So

$$w^{f_0(s)} = \mathrm{Acc}(\mathcal{D}) \quad \text{where } f_0(X) := \prod_{d_0 \in \mathcal{D}_0} (X + d_0).$$

1. The Prover computes the witness

$$w := g_1^{\prod_{d \in \mathcal{D} \setminus \mathcal{D}_0}^{(s+d)}}.$$

2. The Prover gets a challenge $\alpha \in \mathbb{F}_p^*$ and computes a polynomial $h_0(X) \in \mathbb{F}_p[X]$ such that

$$f_0(X) = (X + \alpha)h_0(X) + \beta.$$

3. The Prover computes $Q := w^{h_0(X)}$ and sends $\{Q, w, \alpha\}$ to the Verifier along with a non-interactive `PoK` of the polynomial $h(X)$. The Verifier then checks whether $Q^{s+\alpha}w^{\beta} = \mathrm{Acc}(\mathcal{D})$ by verifying the equation

$$\mathbf{e}(Q, g_2^{s+\alpha}) * \mathbf{e}(w^{\beta}, g_2) = \mathbf{e}(\mathrm{Acc}(\mathcal{D}), g_2).$$

As soon as new data $\mathcal{D}_0$ is added, the node that cares about this data stores a proof of membership for $\mathcal{D}_0$. It then listens to the blockchain updates and keeps updating these proofs in response (possibly with computational help from an untrusted server node). If the node goes offline and comes back after some time, the time it will take to update the proofs is linear in the number of transactions that occurred during the time interval.

Dynamic Updates: A user can update a membership proof for his data dynamically by looking at the new accumulated digest. We describe how a user can do so when the data inserted/deleted in a block is data that he **does not** care about.

Let U_0 be a user on the network and $\mathcal{D}_0$ the data he cares about. For efficient updates he needs to store the set

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\}.$$

As before, set

$$f_0(X) := \prod_{d_0 \in \mathcal{D}_0} (X + d).$$

For a new datum $d \in \mathbb{F}_p^*$ added to the accumulator, the user needs to compute

$$\mathbf{new}(w(\mathcal{D}_0)) := w(\mathcal{D}_0)^{s+d}, \;\; \mathbf{new}(Q) := Q^{s+d}$$

Now

$$\mathbf{new}(Q) := Q^{s+d} = Q^{s+\alpha} * Q^{d-\alpha} = Aw^{-\beta} * Q^{d-\alpha}.$$

Furthermore, he updates

$$\mathbf{new}(w(\mathcal{D}_0)^{s^j}) = w(\mathcal{D}_0)^{s^{j+1}} * w(\mathcal{D}_0)^{s^j d} \quad \text{for } j = 1, \cdots, \#\mathcal{D}_0 - 1.$$

The most expensive part of this update is the element $\mathbf{new}(w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}})$. To compute

$$\mathbf{new}(w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}) = w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0+1}} * w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}d},$$

he needs to compute $w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0+1}}$. To this end, he computes the degree $\#\mathcal{D}_0$ polynomial

$$f_1(X) := f_0(X)(X+d) - X^{\#\mathcal{D}_0+1}$$

and computes $w(\mathcal{D}_0)^{f_1(s)}$ using the set

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\}.$$

He then computes

$$w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0+1}} = \mathbf{new}(A) * w(\mathcal{D}_0)^{f_1(s)}.$$

The non-interactive PoKE between $w(\mathcal{D}_0)$ and Q remains unchanged. Once the new set

$$\{\mathbf{new}(w(\mathcal{D}_0)), \mathbf{new}(w(\mathcal{D}_0))^s, \cdots, \mathbf{new}(w(\mathcal{D}_0))^{s^{\#\mathcal{D}_0}}\} \cup \{\mathbf{new}(Q)\}$$

has been obtained and stored, the old set

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\} \cup \{Q\}$$

is discarded. These computations are highly parallelizable.

Proofs of new elements: Let A be the accumulated digest at the end of the last block, $\mathcal{D}_{\mathbf{new}}$ the data inserted in the block and $f(X) := \prod_{d \in \mathcal{D}_{\mathbf{new}}} (X+d)$. The new accumulated digest is then given by $\mathbf{new}(A) := A^{f(s)}$. Let $\mathcal{D}_0 \subseteq \mathcal{D}$ be the subset of this data set that a certain user U_0 cares about and set $f_0(X) := \prod_{d \in \mathcal{D}_0} (X+d)$. The membership witness of $\mathcal{D}_0$ is given by

$$w_0 := A^{\prod_{d \in \mathcal{D} \setminus \mathcal{D}_0} (s+d)}.$$

Let $\mathcal{D}_1$ be the older data that U_0 cares about, w_1 be the membership witness of $\mathcal{D}_1$ with respect to the state A and set $f_1(X) := \prod_{d \in \mathcal{D}_1} (X+d)$. Then the (updated) witness of $\mathcal{D}_1$ with respect to the state $\mathbf{new}(A)$ is given by

$$A^{\prod_{d \in \mathcal{D} \setminus \mathcal{D}_1} (s+d)}.$$

The witness of $\mathcal{D}_1 \cup \mathcal{D}_0$ with respect to $\mathbf{new}(A)$ is given by

$$w_{0,1} = w_1^{\prod_{d \in \mathcal{D}_{\mathbf{new}} \setminus \mathcal{D}_0} s+d}.$$

The user possesses the set

$$\{w_1, w_1^s, \cdots, w_1^{s^{\#\mathcal{D}_1}}\}$$

which he uses to compute the set

$$\{w_1, w_1^s, \cdots, w_1^{s^{\#\mathcal{D}_1 \cup \mathcal{D}_{\mathbf{new}}}}\}$$

and subsequently, the set

$$\{w_{0,1}, w_{0,1}^s, \cdots, w_{0,1}^{s^{\#\mathcal{D}_1 \cup \mathcal{D}_0}}\}.$$

Deletions: Let $\mathcal{D}_1$ be the data that a user cares about and let $\mathcal{D}_0 \in \mathcal{D}_1$ be the subset deleted in a certain block. Let A be the accumulated digest at the end of the last block, $\mathcal{D}_{\text{del}}$ the data in the new block and $f(X) := \prod_{d \in \mathcal{D}_{\text{del}}} (X + d)$. Then the accumulated digest A_{del} after deleting $\mathcal{D}_{\text{del}}$ is the element of $\mathbb{G}_1$ such that $A_{\text{del}}^{f(s)} = A$.

Let w_1 be the membership witness of $\mathcal{D}_1$ with respect to the state A. Then the witness of $\mathcal{D}_1 \setminus \mathcal{D}_0$ with respect to the state A_{del} is given an element $w_{1,\text{del}} \in \mathbb{G}_1$ such that

$$w_{1,\text{del}}^{\prod_{d \in \mathcal{D}_1 \setminus \mathcal{D}_0} (s+d)} = A_{\text{del}}.$$

So

$$w_{1,\text{del}}^{\prod_{d \in \mathcal{D}_{\text{del}} \setminus \mathcal{D}_0} (s+d)} = w_1.$$

Hence $w_{1,\text{del}}$ can be computed by extracting the $\left(\prod_{d \in \mathcal{D}_{\text{del}} \setminus \mathcal{D}_0} (s+d)\right)$-th root of w_1, the procedure for which was sketched earlier.

Updating proofs in response to deletions: As before, let A be the accumulated digest, $\mathcal{D}_0$ the data the user cares about and $w(\mathcal{D}_0)$ its witness. For efficiently updating his storage in response to a delete, the user will need to store the set

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\} \cup \{Q\}.$$

Let $d \notin \mathcal{D}_0$ be a datum that gets deleted, resulting in the new accumulated digest being

$$\mathbf{new}(A) = A^{(s+d)^{-1}}.$$

Then the user updates his storage as follows. He first computes $h_0(X) \in \mathbb{F}_p[X]$, $\beta_0 \in \mathbb{F}_p$ such that $f_0(X) = (X + d)h(X) + \beta_0$. Now,

$$\mathbf{new}(A) = \mathbf{new}(w)^{f_0(s)} = w^{h_0(s)} * \mathbf{new}(w)^{\beta}.$$

Hence,

$$\mathbf{new}(w) = (\mathbf{new}(A) * w^{h_0(s)})^{{\beta_0}^{-1}} = (\mathbf{new}(A) * w^{h_0(s)})^{\beta_0^{p-2}}.$$

Thus, for $i = 1, \cdots, \#\mathcal{D}_0$, compute $h_i(X) \in \mathbb{F}_p[X]$, $\beta_i \in \mathbb{F}_p$ such that $X^i = (X + d)h_i(X) + \beta_i$. It is easy to see that

$$\beta_i = (-d)^i \in \mathbb{F}_p^*, \;\; h_i(X) = \sum_{j=0}^{i-1} (-d)^{i-1-j} X^j.$$

Now,

$$\mathbf{new}(w)^{s^i} = \mathbf{new}(w)^{\beta_i} * w^{h_i(s)}, \;\; i = 1, \cdots, \#\mathcal{D}_0$$

and

$$\mathbf{new}(Q) = \mathbf{new}(w)^{h(s)}.$$

These computations are highly parallelizable. Once the new set

$$\{\mathbf{new}(w(\mathcal{D}_0)), \mathbf{new}(w(\mathcal{D}_0))^s, \cdots, \mathbf{new}(w(\mathcal{D}_0))^{s^{\#\mathcal{D}_0}}\} \cup \{\mathbf{new}(Q)\}$$

has been obtained and stored, the old set

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\} \cup \{Q\}$$

is discarded.

Possible speed-up: Let $\mathcal{D}_1, \cdots, \mathcal{D}_r$ be the data sets associated to different users that the validators need to delete. Let w_i be the witness for $\mathcal{D}_i$ and $f_i(X) := \prod_{d \in \mathcal{D}_i} (X+d)$. Then $w_i^{f_i(s)} = A$, the accumulated digest at the end of the last block. As discussed earlier, the users provide elements Q_i such that $Q_i^{s+\alpha_i} w_i^{\beta_i} = A$ where $\alpha_i \in \mathbb{F}_p^*$ is a challenge generated by a (pre-agreed upon) hashing algorithm and $\beta_i := f_i(X) \pmod{X + \alpha_i}$. Instead of verifying the r equations

$$Q_i^{s+\alpha_i} w_i^{\beta_i} = A, \;\; i = 1, \cdots, r$$

separately, it would be faster to verify the single equation

$$\prod_{i=1}^{r} (Q_i^{s+\alpha_i} w_i^{\beta_i})^{a_i} = A^{\sum_{i=1}^r a_i}$$

where $a_1, \cdots, a_r \in \mathbb{F}_p^*$ are randomly generated. This is analogous to a trick introduced in the Bulletproofs paper ([BBB+18]).

The flip side is that if this equation fails to hold, there is no choice but to fall back on verifying the r equations individually. But in a **permissioned** blockchain setting where false proofs are relatively rare, this might be a good way to speed up the verification.

Dealing with the non-availability of a witness: Suppose a quorum of processing validators need to delete a data set $\mathcal{D}_0$. To do so, they need the witness $w(\mathcal{D}_0)$ for this data set so they can update the accumulated digest from A to $w(\mathcal{D}_0)$. In this model, the user who cares about the data is repsonsible for storing it and maintaining the witness by listening to the blockchain and updating the proof in reponse. So, if the user is offline or is unable to provide a witness, there is no efficient way to delete the data set.

In this event, we propose that the deletion of $\mathcal{D}_0$ is postponed until the witness is available. Instead, the validators could insert the set

$$-\mathcal{D}_0 := \{-d_0 \;:\; d_0 \in \mathcal{D}_0\}$$

and use a membership proof for $\mathcal{D}_0$ to record the fact that $\mathcal{D}_0$ is to be deleted.

Non-membership witnesses: Let $\mathcal{D}_0$ be a set disjoint from the accumulated set $\mathcal{D}$. Write

$$f_0(X) := \prod_{d \in \mathcal{D}_0} (X + d).$$

1. The Prover computes $\bar{w}(\mathcal{D}_0)$ and a polynomial $h(X) \in \mathbb{F}_p[X]$ relatively prime to $f_0(X)$ and of degree $\leq \#\mathcal{D}_0 - 1$ such that

$$\bar{w}(\mathcal{D}_0)^{f_0(s)} * A = g_1^{h(s)}.$$

2. The Prover computes polynomials $\bar{h}(X), q(X) \in \mathbb{F}_p[X]$ of degree $\leq \#\mathcal{D}_0 - 1$ such that

$$h(X)\bar{h}(X) - 1 = f_0(X)q(X).$$

3. The Prover computes $g_1^{h(s)}, g_2^{\bar{h}(s)}, g_1^{q(s)}$.
4. The Prover sends
$$\bar{w}(\mathcal{D}_0), g_1^{h(s)}, g_2^{\bar{h}(s)}, g_1^{q(s)}$$
along with non-interactive PoKs for $h(X), \bar{h}(X), q(X)$.
5. The Verifier checks the non-interactive PoKs. He then checks whether
$$\mathbf{e}(\bar{w}(\mathcal{D}_0), g_2^{f_0(s)}) * \mathbf{e}(A, g_2) = \mathbf{e}(g_1^{h(s)}, g_2)$$
and
$$\mathbf{e}(g_1^{h(s)}, g_2^{\bar{h}(s)}) = \mathbf{e}(g_1^{q(s)}, g_2^{f_0(s)}) * \mathbf{e}(g_1, g_2).$$

Updates: With notations as above, suppose a set $\mathcal{D}_1$ is added to the accumulated set. Set
$$f_1(X) := \prod_{d \in \mathcal{D}_1} (X + d).$$
Then $\mathbf{new}(A) = A^{f_1(s)}$ and
$$\bar{w}(\mathcal{D}_0)^{f_0(s)f_1(s)} * \mathbf{new}(A) = g_1^{h(s)f_1(s)}.$$
Write
$$h(X)f_1(X) = f_0(X)e_1(X) + h_1(X), \ \ \deg h_1(X) < \#\mathcal{D}_0.$$
Set $\mathbf{new}(\bar{w}(\mathcal{D}_0)) := \bar{w}(\mathcal{D}_0)^{f_1(s)} g_1^{-e_1(s)}$. Then
$$\mathbf{new}(\bar{w}(\mathcal{D}_0))^{f_0(s)} * \mathbf{new}(A) = g_1^{h_1(s)}.$$
Compute polynomials $\bar{h}_1(X), q_1(X) \in \mathbb{F}_p[X]$ of degree $\leq \#\mathcal{D}_0 - 1$ such that
$$h_1(X)\bar{h}_1(X) - 1 = f_0(X)q_1(X).$$
Compute $g_1^{h_1(s)}, g_2^{\bar{h}_1(s)}, g_1^{q_1(s)}$.
Note that
$$\mathbf{new}(h(X)) = h(X)(X + d) \pmod{f_0(X)}, \ \ \mathbf{new}(\bar{h}(X)) = \bar{h}(X)(X + d)^{-1} \pmod{f_0(X)}.$$

The run time for this update is $\mathbf{O}(\#\mathcal{D}_1 * \#\mathcal{D}_0 * \log(\#\mathcal{D}_0))$. In particular, it is independent of the size of the accumulator.

Remark: Our non-membership proof consists of seven group elements. It would be nice if we could compress the size of this set. Far more importantly, the user may need to store and update the polynomial $h(X)$ which is of degree $< \#\mathcal{D}_0$. Without knowledge of the polynomial, the updates would be more difficult. That being said, the user will usually not need to communicate the polynomial to a Verifier, relying instead on a non-interactive PoKE as described above.

3 Membership verification for a node that goes offline

Suppose a node goes offline or fails to update its membership witness for a data set $\mathcal{D}_0$. Let A_0 be the accumulated digest with respect to which the user has a membership proof for $\mathcal{D}_0$. Let $A_1, \cdots, A_N$ denote the accumulated digests at the end of the blocks since the time the user stopped updating the membership proof. To compute the membership proof with espect to the most recent accumulated digest A_N, the user must go through the process which has a run time linear in the number of elements inserted/deleted during this time. But to verify that $\mathcal{D}_0$ has not been deleted is a bit easier. It is possible to verify that a membership witness for $\mathcal{D}_0$ exists without actually computing this witness.

For each integer i, let $A_{i,\text{del}}$ denote the accumulated digest resulting from the batched deletes in that block. So the accumulated digests occur in the sequence

$$A_0 \longrightarrow A_{0,\text{del}} \longrightarrow A_1 \longrightarrow \cdots \longrightarrow A_{N-1,\text{del}} \longrightarrow A_N.$$

The oracle nodes store non-interactive `PoK`s for the logarithm between $A_{i,\text{del}}$ and A_i and that between $A_{i,\text{del}}$ and A_{i+1}. Each of these proofs is given by a single element of $\mathbb{G}_2$.

Let $h_i \in \mathbb{G}_2$ denote the non-interactive `PoK` for the logarithm between $A_{i,\text{del}}$ and A_{i+1}. In other words, we have

$$\mathbf{e}(A_{i,\text{del}}, h_i) = \mathbf{e}(A_{i+1}, g_2).$$

The user verifies the equation

$$\prod_{i=0}^{N-1} \mathbf{e}(A_{i,\text{del}}, h_i) = \mathbf{e}(\prod_{i=0}^{N-1} A_i, g_2).$$

This verification is sufficient to convince the user that every A_{i+1} was obtained from $A_{i,\text{del}}$ by a series of insertions. So, to verify that none of the elements of $\mathcal{D}_0$ were deleted between the states A_0 and A_N, it suffices to check that none of these elements were deleted during the transitions $A_i \longrightarrow A_{i,\text{del}}$. This is the more arduous part and will require non-membership proofs, the computation of which have a run time complexities linear in the size of the set deleted in each block and the user will need access to the s-powers stored by the oracle nodes. However these proofs are highly parallelizable.

4 Summary of the storage and the tasks of the nodes

The user's storage: Let U_0 be a user on the network and let $\mathcal{D}_0$ be the data he cares about. The user stores the following:

- the data $\mathcal{D}_0$
- the most recent accumulated digest
- the four points g_1, g_1^s, g_2, g_2^s
- the membership witness $w(\mathcal{D}_0)$ with respect to the most recent accumulated digest
- the non-membership $\bar{w}(-\mathcal{D}_0)$ for the set $-\mathcal{D}_0 := \{-d : \ d \in -\mathcal{D}_0\}$ with respect to the most recent accumulated digest.
- the sets

$$\{g_1, g_1^s, \cdots, g_1^{s^{\#\mathcal{D}_0}}\}, \ \ \{g_2, g_2^s, \cdots, g_2^{s^{\#\mathcal{D}_0}}\}$$

- the sets

$$\{w(\mathcal{D}_0), w(\mathcal{D}_0)^s, \cdots, w(\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\}, \ \ \{\bar{w}(-\mathcal{D}_0), \bar{w}(-\mathcal{D}_0)^s, \cdots, \bar{w}(-\mathcal{D}_0)^{s^{\#\mathcal{D}_0}}\}.$$

These sets are efficiently updatable and allow the user to compute a membership proof for any subset of $\mathcal{D}_0$ or a non-membership proof for any subset of $-\mathcal{D}_0$.

Thus, the user's storage is linear in the number of data elements he cares about and is otherwise independent of the size of the blockchain. However, the user must continuously listen to the blockchain and keep updating his witnesses. If the node goes offline and returns after a while, it needs to know the intermediate accumulated digests in the interim period in addition to the elements that were added/deleted during this time.

Hence, we need a server node to record the sets

$$\{A, A^s, \cdots, A^{s^{\#\text{block size}}}\}$$

where A is the accumulated digest at the end of the block. It is desirable for this node to lie outside the critical consensus path, i.e. it is preferable for it not to be a processing validator. The server node is trusted for availability but not necessarily for integrity since false data would be incompatible with the most recent block header. While requiring the server node to store the intermediate accumulated digests means more storage for this node (linear in the number of accumulated data elements), it allows the users to perform dynamic stateless updates when they go offline and return after a while. The alternative would mean more storage for the users or more interaction between the users and the validators (both of which we would like to avoid like the plague).

The validator's storage: Since the processing validators use the distributed private key for accumulation of data, they need not store the public parameter they generate. But the public parameter must be stored by some server node preferably outside the consensus path. A *processing validator* V_i stores the following:

• His part s_i of the secret key.
• It would also be beneficial to store the set $\{s_i^2, \cdots, s_i^{\text{block size}}\} \subseteq \mathbb{F}_p^*$ rather than computing it on the fly each time. This would help speed up the computations involved in processing new transactions and would still allow for a constant amount of storage.
• His portions of the other parts of the secret keys that were distributed with the Shamir key-sharing scheme
• the most recent accumulated digest
• the four points g_1, g_1^s, g_2, g_2^s
• The smart contract code (in the intermediate version). One of our primarily goals is to move to a design where the processing validators do not need to hold the smart contract code either.

A *verifying validator* stores the following:

• the most recent accumulated digest
• the four points g_1, g_1^s, g_2, g_2^s

The validator's tasks: The *processing* validators need to update the accumulated digest in response to the new (approved) transactions. For each block, the validators get the data sets $\mathcal{D}_{\text{del}}$ and $\mathcal{D}_{\text{add}}$ to be deleted and added respectively. They proceed as follows.

1. They receive (from the clients or from an aggregator) the membership witnesses for the data sets that constitute $\mathcal{D}_{\text{del}}$. Using Shamir's aggregation trick and with the distributed private key, they compute the membership witness

$$A_{\text{del}} := A^{\prod_{d \in \mathcal{D}_{\text{del}}} (s+d)^{-1}}$$

of $\mathcal{D}_{\text{del}}$. This is the accumulated digest after $\mathcal{D}_{\text{del}}$ has been deleted. They compute and broadcast a non-interactive `PoE` for the equation

$$A_{\text{del}}^{\prod_{d \in \mathcal{D}_{\text{del}}} (s+d)} = A.$$

2. They compute

$$A_{\mathbf{new}} := A_{\text{del}}^{\prod_{d \in \mathcal{D}_{\text{del}}} (s+d)}.$$

They compute and broadcast a a non-interactive `PoE` for this equation.

The processing validators **might also need to help users update their membership proofs when the set of data elements they care about undergoes an update** (additions/deletions). The users whose data remains unchanged can update their proofs without any interaction with the validators.

The *verifying* validators participate in consensus by verifying that the computations involved are done correctly. Instead of performing the same computations themselves, they merely check the succinct `PoE` that the processing validators send them.

The user's tasks: The user needs to maintain and update the proof(s) for the data elements he cares about. He could outsource his computation to an **untrusted** third party, relying on a succinct `PoE` for verification.

Proofs of Retrievability: A key difference between this proposal and what we currently have in place at the moment is that we arte proposing the inclusion of an untrusted server node that stores all of the data. If the server node falsifies data, it would be incompatible with the latest block header that is broadcasted to all nodes. However, it is desirable for each node to be able to ensure that the data it cares about is being stored. To this end, we propose using a publicly verifiable proof of retrievability scheme such as that constructed in [SW08].

5 Vector Commitments

The aim of this section is to construct a Vector Commitment with constant sized openings using the universal bilinear accumulator constructed in the preceding section. Informally, a Vector Commitment is a binding commitment to a vector in the same way that an accumulator is a binding commitment to a set.

The first Vector Commitment with public parameters as well as openings of constant size was constructed in [BBF19] using their universal group-based accumulator. Unfortunately, this does not seem feasible for a bilinear Vector Commitment since the bilinear accumulator has linear public parameters. But our construction does yield a bilinear VC with linear public parameters and openings of constant size which we expect to have a significant speed advantage over a group-based VC. Furthermore, rather than storing the entire public parameter, the Verifier only needs to store the set $\{g_1, g_1^s, g_2, g_2^s\}$ in order to verify membership or non-membership proofs for data sets. Thus, his total amount of necessary storage is of constant size.

Definition 5.1. A *Vector Commitment* (VC) is a tuple consisting of the following PPT algorithms:
1. `VC.Setup`$(\lambda, n, \mathcal{M})$: Given security parameter λ, length n of the vector and message space $\mathcal{M}$ of vector components, output public parameters `pp` which are implicit inputs to all the following algorithms.

2. `VC.Com`($\mathbf{m}$) $\longrightarrow \tau$: Given an input $\mathbf{m} = (m_1, \cdots, m_n)$ output a commitment com.

3. `VC.Update`(com, m, i, τ): Given an input message m and a position i, output a commitment com and advice τ.

4. `VC.Open`(com, m, i, τ): On input $m \in \mathcal{M}$ and $i \in [1, n]$, the commitment com and advice τ, output an opening π that proves m is the i-th committed element of com.

5. `VC.Verify`(com, m, i, τ)$\longrightarrow$ 0/1: On input commitment com, an index $i \in [1, n]$ and an opening proof π, output 1 (accept) or 0 (reject).

A vector commitment is said to be a *subvector commitment* (SVC) if given a vector $\mathbf{m}$ and a subvector $\mathbf{m}'$, the committer may open the commitments at all the positions of m' simultaneously. This notion was first introduced in [LM18]. It is necessary for each opening to be of size independent of the length of $\mathbf{m}'$, since otherwise it would be no more efficient than opening the positions separately. For instance, a Merkle tree is an example of a Vector Commitment that is not a subvector commitment since its position openings are not constant sized and secondly, the openings of several positions cannot be compressed into a single proof. In the rest of this section, we construct a SVC using the accumulator constructed in Section 2.

We start by constructing a bilinear accumulator as in the last section. The message space $\mathcal{M}$ is the set $\{0,1\}^*$. Our construction associates the element $i + p\mathbb{Z} \in \mathbb{F}_p^*$ for each index i of the vector. We now define a bit-vector $\mathbf{m} = (m_1, \cdots, m_{p-1})$ of length $p-1$ as follows. For each index i, we set

$$m_j = \begin{cases} 1 & \text{if } j + p\mathbb{Z} \text{ was accumulated .} \\ 0 & \text{otherwise.} \end{cases}$$

The bit-vector $\mathbf{m}$ is *sparse*, i.e. most of its entries are 0. The opening of the i-th index is a membership proof of $i + p\mathbb{Z}$ if $m_i = 1$ and a non-membership proof if $m_i = 0$. With the accumulator we constructed in the last section, each opening is of constant size. Furthermore, the openings of multiple indices can be batched into a constant sized proof by aggregating all the membership witnesses for $\mathbb{F}_p^*$-elements on the indices opened to 1 and batching all the non-membership witnesses for elements at the indices opened to 0.

We use our accumulator to commit to the set of elements corresponding to indices such that $m_i = 1$. The opening of the i-th index to m_i is an inclusion proof for d_i and the opening to $m_i = 0$ is an exclusion proof for d_i. With our bilinear accumulator, the opening of each index is constant-sized. Furthermore, the openings of multiple indices can be batched into a single constant sized proof using membership proofs for elements on the indices opened to elements of $\mathbb{F}_p^*$ and non-membership proofs for elements opened to 0.

A key-value map commitment: Following the ideas in [BBF19], we use our VC to build a key-value map as follows. The key-space is represented by the positions in the sparse vector. The associated value is the data at the key's position. The complexity of the commitment is proportional to the number of bit indices set to 1 and hence, is independent of the length of the vector.

For a datum **datum** represented by a string, we have a hashing algorithm that maps **datum** to an element of $\mathbb{F}_p^*$, i.e. some non-zero integer $n \pmod p$. The key/address of this datum will be the integer n represented in the binary form. Conversely, for a key **key** in the form of a binary integer representation, we acquire the integer n that **key** represents. The element $n \pmod p \in \mathbb{F}_p^*$ is the value corresponding to the key **key**. Thus, the key-value pair is substantially simpler than that of a MPT or the one arising from an accumulator based on hidden order groups.

References

[BBB+18] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, *Bulletproofs: Short proofs for confidential transactions and more.*

[BBF19] D. Boneh, B. Bunz, B. Fisch, *Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains*

[BGG17] S. Bowe, A. Gabizon, M. Green, *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*

[BGM17] S. Bowe, A. Gabizon, I. Myers, *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*

[CF13] D. Catalano, D. Fiore, *Vector commitments and their applications*

[Ngu05] L. Nguyen, *Accumulators from bilinear pairings and applications*, CT-RSA, 3376:275–292, 2005.

[SW08] H. Shacham, B. Waters, *Compact Proofs of Retrievability*

[T19] S. Thakur, *Batching non-membership proofs with bilinear accumulators*, Preprint

[Wes18] B. Wesolowski, *Efficient verifiable delay functions*, Cryptology ePrint Archive, Report 2018/623, 2018. https://eprint.iacr.org/2018/623.