Snarkifying STMs and ALBA-BLS

Raphael Toledo

December 2024

1 Introduction

This is a short document on snarkifying STMs as well as ALBA-BLS, that is the combination of Alba and Mithril (BLS signatures) to obtain a scheme with better performances for the Peras and Leios projects.

1.1 Notations

We denote by $\leftarrow$ the assignment and by $\xleftarrow{\$}$ the uniform sampling from a definition, e.g. $r \xleftarrow{\$} \mathbb{Z}_p$ is an element taken at uniformly random of $\mathbb{Z}_p$.

Let $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, g_T, e(\cdot,\cdot))$ be a secure asymmetric bilinear group where g_1, g_2, g_T are generators of the groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ of prime order p and $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an efficient map such that $e(g_1, g_2) = g_T$. For readability, we may omit $\mathcal{G}$ from the algorithms' interface.

Let $\mathrm{H}_{\mathcal{X}} : \{0,1\}^* \leftarrow \mathcal{X}$ an efficient hash function which outputs an element of set $\mathcal{X}$, be it $\mathbb{Z}_p$ or $\mathbb{G}_x$ where $x \in \{1, 2, T\}$. We may abuse notations and define H as outputting elements in $\{0,1\}^k$ with k depending on the context or use as input of H the binary representation of an integer or group element.

The function $\mathsf{Card}(\mathcal{X})$ returns the cardinality of the set $\mathcal{X}$.

The symbol == is an equality check.

2 Cryptographic Primitives

2.1 Signature schemes

A signature scheme is a tuple of algorithm $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$

- $(\mathrm{sk}, \mathrm{vk}) \xleftarrow{\$} \mathsf{KeyGen}(\mathcal{G})$: Sample the private key sk and corresponding public key vk and return $(\mathrm{sk}, \mathrm{vk})$
- $\sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, \mathrm{msg})$: Generate the signature σ on message msg using secret key sk.
- $0/1 \leftarrow \mathsf{Verify}(\mathrm{vk}, \sigma, \mathrm{msg})$: Return 1 if σ is a valid signature on message msg using verification key vk, otherwise 0.

2.1.1 BLS signature

The BLS signature scheme is a tuple of algorithm $\mathsf{BLS}.(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ defined over Let the bilinear group $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, g_T, e(\cdot,\cdot))$ defined as follows.

- $(\mathrm{sk}, \mathrm{vk}) \xleftarrow{\$} \mathsf{KeyGen}(\mathcal{G})$: Sample $\mathrm{sk} \xleftarrow{\$} \mathbb{Z}_p$, and return $(\mathrm{sk}, {g_2}^{\mathrm{sk}})$
- $\sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, m)$: Return $\mathrm{H}_{\mathbb{G}_1}(m)^{\mathrm{sk}}$
- $0/1 \leftarrow \mathsf{Verify}(\mathrm{vk}, \sigma, m)$: Return 1 if $e(\mathrm{H}_{\mathbb{G}_1}(m), \mathrm{vk}) = e(\sigma, g_2)$, otherwise 0.

Remark. Note that we can inverse the groups for the verification key, the signature and message.

Remark. We may batch the verification of several signatures for the same message using trusted keys, by checking that $e(\mathrm{H}_{\mathbb{G}_1}(m), \Pi_i\ \mathrm{vk}_i) = e(\Pi_i\ \sigma_i, g_2)$. We may call $\mu = \Pi_i\ \sigma_i$ the *aggregated* signature and similarly $\mathrm{ivk} = \Pi_i\ \mathrm{vk}_i$ the *aggregated* key.

Using BLS signatures with the BLS12-381 elliptic curve, the verification key is 192 byte long (96 when compressed) and the signature 96 byte long (48 when compressed).

2.2 Cryptographic accumulators

We are succinctly presenting here static accumulators. We do not require proofs of non-membership.

A static Dimitar: dynamic? cryptographic accumulator is composed of the following algorithms $\mathsf{Acc}.(\mathsf{Setup}, \mathsf{Add}, \mathsf{Commit}, \mathsf{Prove}, \mathsf{Verify})$:

- $(\mathrm{acc}_0, \mathrm{params}_{\mathrm{acc}}) \leftarrow \mathsf{Setup}(1^\lambda)$: Initialize the accumulator acc_0 and outputs the accumulator parameters $\mathrm{params}_{\mathrm{acc}}$ that we may omit from now on for readability.
- $(\mathrm{acc}', \mathrm{aux}) \leftarrow \mathsf{Add}(\mathrm{acc}, x)$: Add the element x to the accumulator, returning the newly updated accumulated value and auxiliary information, e.g. the index of the element for ordered accumulator.
- $C_{\mathrm{acc}} \leftarrow \mathsf{Commit}(\mathrm{acc})$: Returns the accumulator commitment.
- $\pi_{\mathrm{acc},x} \leftarrow \mathsf{Prove}(\mathrm{acc}, x, \mathrm{aux})$: Returns a proof of membership of the element x in accumulator acc with auxiliary information aux if x is in the accumulator, otherwise $\perp$
- $0/1 \leftarrow \mathsf{Verify}(C_{\mathrm{acc}}, x, \pi_{\mathrm{acc},x})$: Returns 1 if the proof of membership of x in the accumulator acc verifies, otherwise 0.

Remark. We could also consider static accumulators. In that case, there is no Add nor Commit algorithms, and the Setup algorithm also takes as input all elements to accumulate and returns as well the accumulator commitment.

Some accumulators also provide a batch verification algorithm that we define as follow.

- $0/1 \leftarrow \mathsf{AggVerify}(C_{\text{acc}}, \{\pi_{\text{acc},x_i}, x_i\}_i)$

2.2.1 Merkle tree

A Merkle tree is an ordered accumulator that does not present aggregate verification. It is a binary tree that uses recursive hashing of its leaves to obtain a succinct representation of its element, called the tree root.

We are looking here at the use of a Merkle tree in the context of key registration. As such, we define the following algorithms Merkle.(Setup, Add, Commit, Prove, Verify):

- $(\text{acc}, \text{params}_{\text{acc}}]) \leftarrow \mathsf{Setup}(1^\lambda)$: Return the empty set $\emptyset$ and outputs as parameteres the number of elements 0.
- $(\text{acc}', \text{aux}) \leftarrow \mathsf{Add}(\text{acc}, x)$: Add the element x to the set at the position $i = \text{params}_{\text{acc}}$, $\text{acc}' = \text{acc} \bigcup \{x\}$, increment the parameters $\text{params}_{\text{acc}} += 1$ and return $\text{acc}', \mathsf{Card}(\text{acc}')$.
- $C_{\text{acc}} \leftarrow \mathsf{Commit}(\text{acc})$: Compute a Merkle tree of depth $\log_2(\text{params}_{\text{acc}})$ with ordered leaves the elements of acc, recursively hash the leaves and returns the tree root.
- $\pi_{\text{acc},x} \leftarrow \mathsf{Prove}(\text{acc}, x, \text{aux})$: If x is the i^{th} element of acc, return its Merkle path, otherwise $\perp$.
- $0/1 \leftarrow \mathsf{Verify}(C_{\text{acc}}, x, \pi_{\text{acc},x})$: Parse $\pi_{\text{acc},x}$ as the tuple of Merkle nodes $(y_0, \ldots, y_{\log_2(\text{params}_{\text{acc}})})$ and return $\text{H}(y_n \,||\, \text{H}(y_{n-1} \,||\, \text{H}(...\text{H}(y_0||x)...)) == C_{\text{acc}}$.

2.2.2 Pairing-based accumulator

A pairing based accumulator is an accumulator defined on a bilinear group $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, g_T, e(\cdot,\cdot))$ supporting batch verification. These accumulators usually rely on Common Reference Strings (CRS) and thus are bounded. We present here a shorter version of [4], itself being a variant of Nguyen [3] original scheme composed of the algorithms $\mathsf{Acc}_{\mathsf{pair}}$.(Setup, Add, Commit, Prove, Verify, AggVerify).

- $(\text{acc}, \text{params}_{\text{acc}}) \leftarrow \mathsf{Setup}(1^\lambda)$: Let $T = ((g_1, g_1^\tau, \ldots, g_1^{\tau^n}), (g_2, g_2^\tau, \ldots, g_2^{\tau^n}))$ be the n powers of τ of $\mathcal{G}$ where n is the maximum capacity of the accumulator. Return $(\text{acc} = \emptyset, \text{params}_{\text{acc}} = T)$
- $(\text{acc}', \_) \leftarrow \mathsf{Add}(\text{acc}, x)$: Add the scalar element x to the set acc, $\text{acc}' = \text{acc} \bigcup \{x\}$ and return it.

- $C_{\text{acc}} \leftarrow \mathsf{Commit}(\text{acc})$: Return $\Pi_{x_i \in \text{acc}}\ g_1^{(\tau+x_i)}$.
- $\pi_{\text{acc},x} \leftarrow \mathsf{Prove}(\text{acc}, x, \_)$: If $x \in \text{acc}$, return $\Pi_{x_i \in \text{acc}, x_i \neq x}\ g_1^{(\tau+x_i)}$ else $\perp$.
- $0/1 \leftarrow \mathsf{Verify}(C_{\text{acc}}, x, \pi_{\text{acc},x})$: Return $e(\pi_{\text{acc},x}, g_2^x \cdot g_2^\tau) == e(C_{\text{acc}}, g_2)$
- $0/1 \leftarrow \mathsf{AggVerify}(C_{\text{acc}}, \{\pi_{\text{acc},x_i}, x_i\}_n)$: Sample $\gamma \xleftarrow{\$} \mathbb{Z}_p$. Let $\Gamma = \sum_i \gamma^i$. Compute the aggregated $\pi_{\text{agg}} = \Pi_i \gamma^i \pi_{\text{acc},x_i}$ and batched proofs $\pi_{\text{batch}} = \Pi_i \gamma^i \pi_{\text{acc},x_i}^{x_i}$ and return $e(\pi_{\text{batch}}, g_2) \cdot e(\pi_{\text{agg}}, g_2^\tau) == e(C_{\text{acc}}, g_2^\Gamma)$.

2.2.3 KZG Polynomial Commitment variation as accumulator

A KZG polynomial commitment is defined on a bilinear group $\mathcal{G} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, g_T, e(\cdot,\cdot))$ supporting batch verification. To use this commitment as accumulator, we shall work on the interpolation polynomial however: we will want to prove that a polynomial commitment evaluate to our element in a given point.

We define the KZG accumulator as the tuple of algorithms $\mathsf{Acc}_{\mathsf{KZG}}.(\mathsf{Setup}, \mathsf{Add}, \mathsf{Commit}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{AggVerify})$:

- $(\text{acc}, \text{params}_{\text{acc}}) \leftarrow \mathsf{Setup}(1^\lambda)$: Let $T = ((g_1, g_1^\tau, \ldots, g_1^{\tau^n}), (g_2, g_2^\tau))$ be the n powers of τ of $\mathcal{G}$ where n is the maximum capacity of the accumulator. Let $S = \{1, \omega, \cdots \omega^{n-1}\}$ the set of n roots of unity (we could have chosen any distinct elements, but this set has advantages when using SNARKs). Return $(\text{acc} = \emptyset, \text{params}_{\text{acc}} = (T, S))$
- $(\text{acc}', i) \leftarrow \mathsf{Add}(\text{acc}, x)$: If $\mathsf{Card}(\text{acc}) < n$, add the element x to the set acc, $\text{acc}' = \text{acc} \bigcup \{x\}$ and returns it.
- $C_{\text{acc}} \leftarrow \mathsf{Commit}(\text{acc})$: Let $f(X) = \sum_i a_i X^i = \text{ifft}(S, \text{acc})$ where $\text{ifft}(X, Y)$ is the interpolation of coordinates (x_i, y_i). Return $f(g_1^\tau)$.
- $\pi_{\text{acc},x} \leftarrow \mathsf{Prove}(\text{acc}, x, i)$: If $x \in \text{acc}$, let i the position of x in acc, let $h(X) = \frac{f(X)-x}{X-\omega^i}$ with $f(X) = \text{ifft}(S, \text{acc})$, return $(i, h(g_1^\tau))$ else $\perp$.
- $0/1 \leftarrow \mathsf{Verify}(C_{\text{acc}}, x, \pi_{\text{acc},x})$: Format $\pi_{\text{acc},x}$ as (i, π_x) and return the pairing check $e(C_{\text{acc}} - g_1^x, g_2) \cdot e(-\pi_x, g_2^{\tau-\omega^i}) == 1$
- $0/1 \leftarrow \mathsf{AggVerify}(C_{\text{acc}}, \{\pi_{\text{acc},x_i}, x_i\}_n)$: Sample the random coin $\gamma \xleftarrow{\$} \mathbb{Z}_p$ and compute the aggregated coins $\Gamma = \sum_i \gamma^i$. Return the pairing check $e(C_{\text{acc}}^\Gamma - g_1^{\sum_i \gamma^i x_i} + \sum_i \pi_{x_i}^{\gamma^i \omega^i}, g_2) \cdot e(-\sum_i \pi_{x_i}^{\gamma^i}, g_2^\tau) == 1$

2.2.4 KES accumulator

TODO: Cache based. The cold keys are stored in cache (can be revoked and changed through an independent system), and the hot keys are provided in the signature with a proof. Raph: Is this an "accumulator"? Does this fit with the interface previously defined?

2.3 Lottery schemes

A signature scheme is a tuple of algorithm $(\mathsf{Prove}, \mathsf{Verify})$

- $(l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i}) \leftarrow \mathsf{Prove}(\text{sk}_i, \text{msg}, \text{stake}_i, \text{stake}, \text{aux}_{\mathsf{Lottery}})$: Compute out of the party's secret key sk_i, message msg, local stake stake_i, total stake stake and some auxiliary information $\text{aux}_{\mathsf{Lottery}}$, the number of winning lottery tickets l_i, the winning ticket numbers indices_i and the proof of eligibility $\pi_{\mathsf{Lottery},i}$.
- $0/1 \leftarrow \mathsf{Verify}(\text{vk}_i, \text{msg}, l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i})$: Return 1 if the proof of eligibility is $\pi_{\mathsf{Lottery},i}$ is valid for the l_i lottery numbres indices_i, and message msg using the verification key vk_i, otherwise 0. Raph: We may need the stakes here too

2.3.1 Alba's Lottery

TODO:

2.3.2 Mithril's Lottery

In the paper, use Dense Mapping function, based on Elligator to return a number from a group element.

In code, compares a hash $\text{ev} \in \{0,1\}^{512}$ cast as integer and mapped to the interval $[0,1]$, $p = \text{ev}/2^{512}$, to a real number $\phi_f \in [0,1]$ $p < 1 - (1-\phi_f)^w$ where $w = \frac{\text{stake}_i}{\text{stake}}$ is the relative stake. TODO:

2.3.3 Sortition

Compares a VRF's hash to the CDT of a binomal distribution, returning a integer in $[0, \text{stake}_i]$. TODO:

3 Certification generation

3.1 High-level Overview

We can model a certification as a protocol with six algorithms:

- $\text{params} \leftarrow \mathsf{Setup}(1^\lambda, \text{threshold}, \text{aux}_{\text{params}})$. Setup the parameters params depending on the security parameter λ, threshold to reach and auxiliary parameters $\text{aux}_{\text{params}}$. We omit params from the following algorithms for readability.
- $(\pi_{\text{acc},i}, \text{aux}_{\text{acc},i}) \leftarrow \mathsf{Register}(\text{reg}_i = (\text{vk}_i, \text{stake}_i))$, each party register their key vk_i and stake stake_i in an accumulator which will be committed, with committed value C_{acc}, once the registration is closed retrieving their proof of membership $\pi_{\text{acc},\text{reg}_i}$. Some auxiliary information $\text{aux}_{\text{acc},i}$ may also be

returned such as the total stake stake or other variables to facilitate the handling of membership proofs.

- $\sigma_i/\perp \leftarrow \mathsf{Vote}(\mathrm{sk}_i, \mathrm{msg}, C_{\mathrm{acc}}, \mathrm{stake}_i, \pi_{\mathrm{acc},i}, \mathrm{aux}_{\mathrm{acc},i})$. The voter first plays locally a lottery using their secret key sk_i and stake stake_i, and the total stake stake to find out how many l_i votes they are entitled to, the numbers of the winning tickets $\mathrm{indices}_i$ and getting a proof of eligibility $\pi_{\mathsf{Lottery},i}$. If the voter has lost the lottery, i.e. $l_i = 0$, the algorithm aborts and returns $\perp$. Otherwise, the algorithm generates a vote σ_i on the message msg using the voter's secret key sk_i, the accumulator commitment C_{acc}, the proof of membership $\pi_{\mathrm{acc},i}$, accumulator auxiliary information $\mathrm{aux}_{\mathrm{acc},i}$, number of winning lottery tickets l_i, the corresponding ticket numbers $\mathrm{indices}_i$, and the proof of eligibility $\pi_{\mathsf{Lottery},i}$. The algorithm returns then the vote σ_i that is meant to be sent together with the verification key vk_i derived from sk_i to the aggregator.

- $0/1 \leftarrow \mathsf{Verify}(\sigma_i, \mathrm{vk}_i, \mathrm{msg})$. The verifier verifies the vote σ_i on message msg with the verification key vk_i and returns 1 if the verification is succesful, otherwise 0.

- $\tau/\perp \leftarrow \mathsf{Aggregate}(\{\sigma_i, \mathrm{vk}_i\}, \mathrm{msg})$, the aggregator after checking each vote, aggregates them in a certificate. The aggregator may optionally compress even more the certificate, for instance by choosing a more succinct format or relying on zero-knowledge proofs.

- $0/1 \leftarrow \mathsf{AggVerify}(\tau, \mathrm{msg})$. After decompressing the certificate if needs be, the verifier verifies the certificate τ on message msg and returns 1 if the verification is successful, otherwise 0.

As we can see, the protocol consists of the following three main structures:

- the Accumulator, this structure comprises all verification keys (and corresponding stakes). It can be committed to a value, and parties can prove and verify memberships of elements in the accumulator.

- the Vote, this structure comprises the signature on a message, as well as the proof of eligibility for voting and a proof of registration, i.e. the proof of membership to an accumulator.

- the Certificate, this structure demonstrates knowledge of a quorum valid votes from registered parties on a specific message.

3.2 Mithril signature

Mithril [2] is a Stake-based Threshold Multi-signature scheme that relies on BLS signature properties that can directly be used for certificate generation and Merkle trees as accumulator. We store the tuple of keys and stakes as leaves of a Merkle tree that we commit on at each epoch. Each voter pass the corresponding Merkle tree root and Merkle paths as proofs of validity when

voting during a given epoch. The main advantage of Merkle tree-based accumulator is its verifying time performance and the succinctness of the tree commitment, a single hash. This scheme, however, is costly for proof generation, as the prover needs an extended witness, and the proof size is logarithmic in the tree nodes. We define Mithril protocol as the tuple of algorithms $\mathsf{Mithril}.(\mathsf{Setup}, \mathsf{Register}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$.

In the setup phase, we generate internal parameters k and m out of the adversarial quorum, the active slot coefficient, the target ratio and the security parameters. The parameter k represents the number of lotteries to win to generate a certificate, i.e. the number of signatures the aggregator will receive, and m is the total number of lotteries done.

In the registration phase, each party enters in a Merkle tree their verification key and stake. Once the registration is closed, each party can get the Merkle tree root C_{acc} and their Merkle proof.

$$(\pi_{\text{acc},i}, \text{aux}_{\text{acc},i}) \leftarrow \mathsf{Mithril.Register}(\text{vk}_i, \text{stake}_i)$$

where $(\_, \text{aux}_i) \leftarrow \mathsf{Merkle.Add}(\text{acc}, \text{reg}_i)$, $C_{\text{acc}} = \mathsf{Merkle.Commit}(\text{acc}_{\text{final}})$ and $\pi_{\text{acc},i} \leftarrow \mathsf{Merkle.Prove}(\text{acc}, \text{reg}_i, i)$ with $\text{reg}_i = (\text{vk}_i, \text{stake}_i)$ and $\text{acc}_{\text{final}}$ being the accumulator after adding the last element.

In the voting phase, each party plays the lotteries and depending on the result outputs a Mithril signature composed of the lottery ticket numbers they have won and a BLS signature on the input message appended to the Merkle tree root, the party's registry, i.e. verification key and stake stored in the tree, the registry's index and finally its Merkle proof. These are then sent to the aggregator.

$$\sigma_{\mathsf{Mithril},i} \leftarrow \mathsf{Mithril.Vote}(\text{sk}_i, \text{msg}, C_{\text{acc}}, \text{stake}_i, \pi_{\text{acc},i}, \text{aux}_{\text{acc},i})$$

where the vote is formatted as $(\sigma_i, \text{acc}_i, \text{lotto}_i) \leftarrow \sigma_{\mathsf{Mithril},i}$. The lottery information $\text{lotto}_i = (l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i})$ is composed of the number of winning tickets l_i, the indices indices_i and proof of eligibility $\pi_{\mathsf{Lottery},i}$ which are generated from the lottery on some message $\text{msg}_{\mathsf{Lottery}}$ composed of a random seed and predeterministic value with the local stake stake_i, total stake stake and the internal parameter m for auxiliary information: $(l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i}) \leftarrow \mathsf{Lottery}(\text{sk}_i, \text{msg}_{\mathsf{Lottery}}, \text{stake}_i, \text{stake}, \text{aux}_{\mathsf{Lottery}} = m)$. The accumulator information $\text{acc}_i = (C_{\text{acc}}, \text{reg}_i = (\text{stake}_i, \text{vk}_i), \pi_{\text{acc},i}, \text{aux}_{\text{acc},i})$ comprises the accumulator committed value C_{acc}, the register i composed of the voter's stake stake_i and verification key vk_i, the proof of membership $\pi_{\text{acc},i}$ and auxiliary information $\text{aux}_{\text{acc},i}$, here the register index i. The signature σ_i is a BLS signature on the message msg and Merkle tree's commitment C_{acc}: $\sigma_i \leftarrow \mathsf{BLS.Sign}(\text{sk}_i, C_{\text{acc}}||m)$ if the eligibility check passes, that is $l_i \neq 0$, otherwise $\perp$.

The vote can be verified by calling $\mathsf{Mithril.Verify}(\sigma_{\mathsf{Mithril},i}, \text{vk}_i, \text{msg})$ which verifies each subcomponent appropriately.

In the aggregation phase, the aggregator, after verifying successfully enough vote individually via the Mithril.Verify algorithm, aggregates the verification keys together as well as the signatures together. They return these together with a proof of eligibility, accumulation and aggregation of these signatures.

$$\tau_{\mathsf{Mithril}} \leftarrow \mathsf{Mithril.Aggregate}(\{\sigma_{\mathsf{Mithril},i}\}_i, \mathrm{msg})$$

where the certificate is formatted as $\tau \leftarrow (\mu, \mathrm{ivk}, \pi_{C_{\mathrm{acc}}})$. The aggregated signature μ is the sum of all BLS signatures $\leftarrow \Pi_i \sigma_i$, and the aggregated key ivk the sum of all verification keys $\mathrm{ivk} \leftarrow \Pi_i \mathrm{vk}_i$ the aggregated key. The poofof eligibility, accumulation and aggregation $\pi_{C_{\mathrm{acc}}}$ can either be computed with a SNARK or simply be the concatenation of all necessary information which is the case implemented in Cardano. Mithril finally uses the Octopus algorithm [1] to compress the Merkle proofs and reformat the concatenated signatures in a dictionary storing as values the corresponding winning lottery numbers.

In the verification phase, after verifying the proof of eligibility, accumulation and aggregation, the verifier verifies the aggregated signature on the root and message with the aggregated key.

$$0/1 \leftarrow \mathsf{Mithril.AggVerify}(\tau, \mathrm{msg})$$

formats τ as the aggregated signature μ, the aggregated verifcation key ivk, and the proof of eligibility, accumulation and aggregation $\pi_{C_{\mathrm{acc}}}$: $(\mu, \mathrm{ivk}, \pi_{C_{\mathrm{acc}}}) = \tau$ and returns $\mathsf{BLS.Verify}(\mathrm{ivk}, \mu, \mathrm{msg} \mid\mid C_{\mathrm{acc}})$ if $\pi_{C_{\mathrm{acc}}}$ verifies otherwise 0.

Remark. Mithril is implemented with $\mathbb{G}_2$ elements for verification keys in Cardano and concatenate all the information as proof of eligibility and aggregation.

Compression. In the context of Leios, we have around 3,000 Stake Pool Operators (SPOs) that may participate in the certificate generation. Were we to register all SPOs, we would build a Merkle tree of depth 12 ($2^{12} = 4,095 > 3000$). Hence, assuming we hash the leaves to 32 bytes, using SHA2 of Blake2 for instance, a single Merkle proof would take 384B.

Due to the bias of the stake distribution, 500 SPOs control more than 80% of the total stake, 1,000 SPOs more than 99% of it. Were we to target 500 (resp. 1,000) voters with the lottery, the collection of Merkle proof would thus take 192kB (resp. 384kB).

However, the Merkle proofs can be batched. Mithril uses the Octopus algorithm [1] which may result in drastic gains as shown in benchmarks done on random data displayed in Table 1. These numbers are to take with caution as they heavily depends on the stake distribution. Furthermore, as signatures may be repeated, the "signature" here refers to the unique set of BLS signatures, the index of the signer in the Merkle tree and the lottery indices for which the signatures are valid.

k	m	n	STM (B)	# σ	σ (B)	π_{Merkle} (B)
2271	19,663	3,000	214,792	929	174,256	40,536
2271	19,663	2,000	161,464	690	134,104	27,360
2271	19,663	1,000	94,504	376	81,368	13,136
2271	19,663	500	65,112	206	52,816	6,720

Table 1: Mithril STM signature size, as well as the number and size of its underlying signatures σ and batched Merkle proof π_{Merkle}, using Blake2b with digest length set to 256 for Mithril parameters set as follows $\lambda = 128$, active slot $f = 0.2$, adversarial stake of 40% and target ratio of 55%.

3.3 Alba-BLS

We are showing here how to generically use Alba proofs on BLS signatures to achieve a certification protocol, that is using Alba proof for aggregating BLS signatures, mirroring the k out of m lottery of Mithril, and any Lottery and and accumulation scheme for key registration. The scheme is composed of the tuple of algorithms Alba-BLS.(Setup, Register, Vote, Verify, Aggregate, AggVerify).

In the setup phase, we generate the Decentralized Alba internal parameters out of the lower bound, that must be greater than the adversarial quorum size, an estimate of the set size and the security parameters.

$$\text{params} \leftarrow \mathsf{Alba\text{-}BLS.Setup}(1^\lambda, n_f, \text{aux}_{\text{params}})$$

where the parameters are composed of both the accumulator's and Alba's params = $(\text{params}_{\text{acc}}, \text{params}_{\mathsf{Alba}})$.

In the registration phase, each party enters in a cryptographic accumulator as register their verification key and stake, $\text{reg}_i = (\text{vk}_i, \text{stake}_i)$. Once the registration is closed, each party can get the accumulator commitment C_{acc} and their proof of membership $\pi_{\text{acc},\text{reg}_i}$.

In the voting phase, each party plays the decentralized lottery Lottery.Prove and depending on the result outputs a BLS signature on the input message appended to the cryptographic accumulator commitment, the party's registry, i.e. verification key and stake, stored in the accumulator, and the proof of membership. These are then sent to the aggregator.

$$\sigma_{\mathsf{Alba\text{-}BLS},i} \leftarrow \mathsf{Alba\text{-}BLS.Vote}(\text{sk}_i, \text{msg}, C_{\text{acc}}, \text{stake}_i, \pi_{\text{acc},i}, \text{aux}_{\text{acc},i})$$

with $\sigma_{\mathsf{BLS},i} \leftarrow (\sigma_i, \text{reg}_i = (\text{vk}_i, \text{stake}_i), \pi_{\text{acc},\text{reg}_i}, l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i})$. The Lottery parameters are computed with with the voter's stake stake_i, total stake stake, any lottery axuiliary information $\text{aux}_{\mathsf{Lottery}}$ which includes a random seed, the voter's private key sk_i and predeterministic message $\text{msg}_{\mathsf{Lottery}}$: $(l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i}) \leftarrow \mathsf{Lottery.Prove}(\text{sk}_i, \text{msg}_{\mathsf{Lottery}}, \text{stake}_i, \text{stake}, \text{aux}_{\mathsf{Lottery}})$. The signature is computed

as a BLS signature on both the accumulator's commitment and the input message: $\sigma_i \leftarrow \mathsf{BLS.Sign}(\mathrm{sk}_i, C_{\mathrm{acc}}||m)$ if the eligibility check passes, otherwise $\perp$. The vote can be verified by calling $\mathsf{Alba\text{-}BLS.Verify}(\sigma_{\mathsf{BLS},\mathrm{reg}_i}, \mathrm{vk}_i, \mathrm{msg})$ which verifies the BLS signature and proof of membership appropriately.

In the aggregation phase, the aggregator, after verifying each vote individually, aggregates the verification keys and signatures together. They return these together with a proof of eligibility and aggregation of these signatures. This proof can be computed with a SNARK or simply a concatenation of all the information.

$$\tau \leftarrow \mathsf{Alba\text{-}BLS.Aggregate}(\{\sigma_{\mathsf{Alba\text{-}BLS},i}\}_i, \mathrm{msg})$$

where $\tau \leftarrow (\mu, \mathrm{ivk}, \pi_{C_{\mathrm{acc}}})$. Let $\{\sigma_i, \mathrm{vk}_i\}_n$ be the set of unique BLS signatures and corresponding verification keys output by Alba.Aggregate algorithm. We define by $\mu \leftarrow \Pi_i \sigma_i$ the aggregated signature, and $\mathrm{ivk} \leftarrow \Pi_i vki$ the aggregated key and π_{acc} is a proof of eligibility, accumulation and Alba aggregation of the BLS signatures.

In the verification phase, after verifying the proof of eligibility, accumulation and Alba aggregation, the verifier recomputes the aggregated verification key and verifies the aggregated signature on the accumulator commitment and message.

$$0/1 \leftarrow \mathsf{Alba\text{-}BLS.AggVerify}(\tau, \mathrm{msg})$$

formats τ as the aggregated signature μ, the aggregated verification key ivk, and the proof of eligibility, accumulation and Alba aggregation $\pi_{C_{\mathrm{acc}}}$: $(\mu, \mathrm{ivk}, \pi_{C_{\mathrm{acc}}}) = \tau$ and returns $\mathsf{BLS.Verify}(\mathrm{ivk}, \mu)$ if $\pi_{C_{\mathrm{acc}}}$ verifies otherwise 0.

4 Snarkification

We are looking here at the viability of snarkifying parts of the protocol, as well as the number of constraints and the building blocks needed to snarkify them either in a recursive manner or in standalone.

4.1 Snarkifying each components

4.1.1 Accumulator

Merkle Trees

Pairing based Accumulator

KZG Polynomial Commitment variant Accumulator

4.1.2 Lottery Scheme

Mithril Lottery

Alba Lottery

Sortition

4.1.3 Signatures

BLS signature

KES signature

4.2 Snarkifying Certificate verification

Let SD_e be the stake distribution at epoch e. We are trying to certify a commitment on message msg, potentially $m = \mathsf{SD}_{e+1}$. To support long messages, we will certify a message commitent and not the message itself. We omit for readability the circuit parameters which comprise the commitments on the selections and permutations polynomials.

Public input.

- C_{msg} a commitment of on the message msg (potentially the new stake distribution SD_{e+1} of epoch $e+1$)
- The commitment on stake distribution of epoch e: C_{SD_e}

Private input. Assuming the certificate is an aggregation of n signatures, we have as private input

- The message msg, for instance the new stake distribution $\text{SD}_{e+1} = \{(vk_i^{(e+1)}, \text{stake}_i^{(e+1)}\}_{n'}$ of epoch $e+1$ composed of n' elements
- The message commitment's proof of opening π_{msg}, for instance the set of n' proofs of opening $\Pi_{\text{acc}} = \{\pi_{\text{acc},\text{reg}_i^{(e+1)}}\}_{n'}$ for all $\text{vk}_i^{(e+1)} \in \text{SD}_{e+1}$
- The n verification keys and stakes: $\text{Reg} = \{(vk_i^{(e)}, \text{stake}_i^{(e)})\}_n$
- The total stake at epoch e: $\text{stake}^{(e)}$ Raph: Not sure this should be private
- The aggregated signature μ on the $C(m)||C_{\text{SD}_e}$
- The aggregated key $\text{ivk}^{(e)}$
- The proof of eligibility and aggregation π_{acc} for the verification keys $\{\text{vk}_i\}_n$ committed in C_{SD_e}

Statement. Three items need to be verified,

1. The message's commitment proof verifies successfully,
   $\mathsf{Commit.Verify}(C_{\text{msg}}, m, \pi_m) = 1$
2. The aggregate signature verifies successfully,
   $\mathsf{BLS.AggVerify}(\text{ivk}, \mu, m||C_{\text{SD}_e})) = 1$
   (For Mithril split variant only, the batched signature verifies successfully,
   $\mathsf{BLS.AggVerify}(\text{ivk}', \mu', m||C_{\text{SD}_e})) = 1$)
3. The proof of eligibility, accumulation and aggregation verifies successfully,
   $CS.\mathsf{Verify}(\pi_{\text{acc}}, \text{Reg}, \text{stake}^{(e)}) = 1$

The verification of the proof of eligibility, accumulation and aggregation, when all the information is simply concatenated, becomes,

- Accumulation:
  - If the accumulation scheme does not support batch verification,
    $\forall(\text{reg}_i, \pi_{\text{acc,reg}_i}) \in \text{Reg} \times \pi_{C_{\text{acc}}},\ \mathsf{Acc.Verify}(C_{\text{acc}}, \text{reg}_i, \pi_{\text{acc,reg}_i}) == 1$
  - Otherwise,
    $\mathsf{Acc.AggVerify}(C_{\text{acc}}, \{\text{reg}_i, \pi_{\text{acc,reg}_i}\}_{\forall(\text{reg}_i, \pi_{\text{acc,reg}_i}) \in \text{Reg} \times \pi_{C_{\text{acc}}},}) == 1$
- Aggregation:
  - Aggregation of the verifcation keys,
    $\text{ivk} = \Pi_{\text{vk}_{i \in \text{Reg}}} \text{vk}_i$
  - Aggregation of the BLS signatures,
    $\mu = \Pi_{\sigma_i \in \pi_{\text{acc}}} \sigma_i$
  - For Alba only, that the correct votes were chosen, $\mathsf{Alba.Verify}(\cdot)$
  - For $\mathsf{Mithril}$ split variant only, the batching of the verification keys and BLS signatures were done correctly, $\text{ivk}' = \Pi_{\text{reg}_i \in \text{Reg}} \text{vk}_i^{\text{stake}_i}, \mu' = \Pi_{\text{stake}_i \in \text{Reg and } \sigma_i \in \pi_{\text{acc}}} \sigma_i^{\text{stake}_i}$
- Eligibility,
  $\forall\{\text{reg}_i, l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i}\} \in \pi_{C_{\text{acc}}},$
  $\mathsf{Lottery.Verify}(\text{vk}_i, \text{msg} = \cdot, l_i, \text{indices}_i, \pi_{\mathsf{Lottery},i}) == 1$

4.3 Recursive Certificate

We are looking here at the chain of certificate on stake distributions defined in Mithril but easily extendable to other certificate generation such as $\mathsf{Alba\text{-}BLS}$. Let SD_e be the stake distribution at epoch e. We are trying to certify SD_{e+1}. As such, the message msg that parties vote on with the $\mathsf{Certificate.Vote}(\cdot)$ algorithm is the commitment of the new stake distribution $\text{msg} = C_{\text{acc},e+1} = \mathsf{Acc.Commit}(\mathsf{SD}_{e+1})$. The accumulator used in the same algorithm however $C_{\text{acc},e}$ is a commitment on the previous stake distribution $C_{\text{acc}} = \mathsf{Acc.Commit}(\mathsf{SD}_e)$.

We could say that a certificate consumes a previous (commitment on) SD_e to certify a new (commitment on) SD_{e+1}.

4.3.1 Recursive SNARK of a certificate

In this case, the main difference with subsection 4.2 is that we need to verify additionally a proof of the previous certificate.

Public input.

- The commitment on the new stake distribution: $C_{\mathsf{SD}_{e+1}}$
- The commitment on the current stake distribution: C_{SD_e}
- The commitment on the previous stake distribution: $C_{\mathrm{SD}_{e-1}}$
- The SNARK proof on this certificate $\pi^{(e)}_{\mathrm{SNARK,\ rec}}$

Private input. Assuming the certificate is an aggregation of n signatures, we have as private input

- The new stake distribution $\mathrm{SD}_{e+1} = \{(vk_i^{(e+1)}, \mathrm{stake}_i^{(e+1)}\}_{n'}$ of epoch $e+1$ composed of n' elements
- The message commitment's proof of opening π_{msg}, for instance the set of n' proofs of opening $\Pi_{\mathrm{acc}} = \{\pi_{\mathrm{acc,reg}_i^{(e+1)}}\}_{n'}$ for all $\mathrm{vk}_i^{(e+1)} \in \mathrm{SD}_{e+1}$
- The n verification keys and stakes: $\mathrm{Reg} = \{(vk_i^{(e)}, \mathrm{stake}_i^{(e)})\}_n$
- The total stake at epoch e: $\mathrm{stake}^{(e)}$ Raph: Not sure this should be private
- The aggregated signature μ on the $C(m)||C_{\mathrm{SD}_e}$
- The aggregated key $\mathrm{ivk}^{(e)}$
- The proof of eligibility and aggregation π_{acc} for the verification keys $\{\mathrm{vk}_i\}_n$ committed in C_{SD_e}

Statement. Four items need to be verified,

1. Either the origin was the epoch before, or the previous certificate is valid $e = 0 \vee \mathrm{CS.}\mathsf{Verify}(\pi_{\mathrm{SNARK,\ rec}}, C_{\mathrm{SD}_e}, C_{\mathrm{SD}_{e-1}}) == 1$
2. The message's commitment proof verifies successfully, $\mathsf{Commit.Verify}(C_{\mathrm{msg}}, m, \pi_m) = 1$
3. The aggregate signature verifies successfully, $\mathsf{BLS.AggVerify}(\mathrm{ivk}, \mu, m||C_{\mathrm{SD}_e}) = 1$
   (For Mithril split variant only, the batched signature verifies successfully, $\mathsf{BLS.AggVerify}(\mathrm{ivk}', \mu', m||C_{\mathrm{SD}_e}) = 1$)
4. The proof of eligibility, accumulation and aggregation verifies successfully, $CS.\mathsf{Verify}(\pi_{\mathrm{acc}}, \mathrm{SD}_e, \mathrm{stake}^{(e)}) = 1$

4.3.2 Recursive SNARK of a proof of a certificate

We split here the proof computation in two steps: we first generate a proof for the current epoch's certificate and then generate a recursive proof out of it. We study this case in case the verification of a certificate and of a proof is not possible, e.g. the proof generation is too long or takes too many constraints.

Public input.

- The commitment on the new stake distribution: $C_{\mathsf{SD}_{e+1}}$
- The commitment on the current stake distribution: C_{SD_e}
- The commitment on the previous stake distribution: $C_{\mathrm{SD}_{e-1}}$
- The recursive SNARK proof on the previsou stake distribution certificate $\pi^{(e)}_{\mathrm{SNARK,\ rec}}$
- The SNARK proof of the current stake distribution $\pi^{(e+1)}_{\mathrm{SNARK}}$ computed according to Section 4.2

Private input. There is no private input.

Statement. Two items need to be verified,

1. Either the origin was the epoch before, or the previous certificate is valid $e = 0 \vee \mathrm{CS.}\mathsf{Verify}(\pi^{(e)}_{\mathrm{SNARK,\ rec}}, C_{\mathrm{SD}_e}, C_{\mathrm{SD}_{e-1}}) == 1$
2. The current certificate proof is valid, $\mathrm{CS.}\mathsf{Verify}(\pi^{(e+1)}_{\mathrm{SNARK,\ rec}}, C_{\mathrm{SD}_{e+1}}, C_{\mathrm{SD}_e}) == 1$

Because we are doing several pairing checks with the same "form", we may batch them in circuit. As the first one may be $\perp$ however, in case the previous epoch was the origin, we would need to replace the underlying values in this case by dummy ones. Let A_1 and A_2 be the $\mathbb{G}_1$ elements in Plonk's pairing check, i.e.

$$e(A, g_2^\tau) \cdot e(B, g_2) == 1_T$$

and f and g the functions to respectively compute them out of a proof. Hence, the statement becomes:

1. If the origin was the epoch before, we return dummy values or the ones computed from previous certificate is valid

$$A_1^{(e)} = g_1 \text{ if } e = 0, \text{ else } f(\pi^{(e)}_{\mathrm{SNARK,\ rec}}) \tag{1}$$

$$A_2^{(e)} = g_1^\tau \text{ if } e = 0, \text{ else } g(\pi^{(e)}_{\mathrm{SNARK,\ rec}}) \tag{2}$$

2. Let $A_1^{(e+1)} = f(\pi_{\text{SNARK}}^{(e+1)})$, $A_2^{(e+1)} = g(\pi_{\text{SNARK}}^{(e+1)})$, and $\gamma = \text{H}_{\mathbb{Z}_p}(A_1^{(e)}, A_2^{(e)}, A_1^{(e+1)}, A_2^{(e+1)})$,

$$e(A_1^{(e)} + \gamma \cdot A_1^{(e+1)}, g_2^{\tau}) \cdot e(A_2^{(e)} + \gamma \cdot A_2^{(e+1)}, g_2) == 1$$

Were we to perform the pairing check outside the circuit, we would return as public values $A_1^{(e)} + \gamma \cdot A_1^{(e+1)}$ and $A_2^{(e)} + \gamma \cdot A_2^{(e+1)}$.

References

[1] Jean-Philippe Aumasson and Guillaume Endignoux. Improving stateless hash-based signatures. In *Cryptographers' Track at the RSA Conference*, pages 219–242. Springer, 2018.

[2] Pyrros Chaidos and Aggelos Kiayias. Mithril: Stake-based threshold multisignatures. In *International Conference on Cryptology and Network Security*, pages 239–263. Springer, 2024.

[3] Lan Nguyen. Accumulators from bilinear pairings and applications. In *Topics in Cryptology–CT-RSA 2005: The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005. Proceedings*, pages 275–292. Springer, 2005.

[4] Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2719–2733, 2022.