

Celo Light Client

1 Introduction

We assume we are given groups of prime order r $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$, and denote by \mathbb{F} the finite field of the same size. We are also given generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$. We write \mathbb{G}_1 and \mathbb{G}_2 additively, and \mathbb{G}_t multiplicatively.

We denote by H a hash function taking as input strings of arbitrary length and outputting elements of \mathbb{G}_2 . We model H as a random oracle in the security proof, mainly as that is needed for security of the BLS signature scheme.

BLS signature scheme:

- The message m is an arbitrary string
- The secret key sk is a uniformly chosen element of \mathbb{F} , and the corresponding public key is $pk := sk \cdot g_1$.
- The signature σ of m under sk is

$$\sigma = \mathbf{Sign}(m, sk) := sk \cdot H(m)$$

BLS has the extremely useful property that $\mathbf{Sign}(m, sk_1 + \dots + sk_t) = \sum_{i \in [t]} \mathbf{Sign}(m, sk_i)$

Thus, a set of users with private keys $\{sk_i\}_{i \in [t]}$ and public keys $\{pk_i\}_{i \in [t]}$ can sign m separately and their signatures can be aggregated to a single signature under public key $pk_{agg} := \sum_{i \in [t]} pk_i$.

Registered key owners: To avoid the so-called “rogue key-attack” when aggregating BLS signatures, we must only allow signatures with public keys pk_i such that a proof of knowledge of sk has been provided. Thus, when a key pk is authorized to participate in committees, such a zk proof of knowledge of sk , e.g. Schnorr, must be provided.

2 The light client protocol

We refer to stake holders by their registered public key. Thus, when we refer to a *committee*, we mean a set of public keys $\{pk\}$.

Denote by C_i the committee of the i 'th epoch.¹ The light client \mathbf{V} will only verify the identity of the current epoch committee. It will satisfy the following completeness and soundness properties

¹We are assuming here an idealized consensus functionality where this value is well-defined; e.g. not dealing with forks when describing the light client.

Completeness (liveness): If all committees up to epoch T have had a $2/3$ -honest majority, then \mathbf{V} will obtain the correct value C_T

Soundness: If all committees $\{C_i\}_{i \in [T]}$ have more than $1/3$ honest players then \mathbf{V} will not be convinced of a wrong value C_T .

Last block header of epoch The structure of the last block header of an epoch is important for the light client protocol; and so we describe some of its details.

1. It will contain a string $\mathbf{m} = (S_1, S_2)$, for the two sets $S_1 = \{\mathbf{pk}_i\}_{i \in [s]}$, $S_2 = \{\mathbf{pk}'_i\}_{i \in [s]}$ of the keys we add and remove from the validator set, i.e. $C_T = (C_{T-1} \setminus S_1) \cup S_2$.
2. It will contain a string $x \in \{0, 1\}^t$ signifying what validators from C_{T-1} signed \mathbf{m} .
3. Let $\mathbf{pk}_{\text{agg}} := \sum_{i \in [t]} \mathbf{pk}_i$, where $C_{T-1} = \{\mathbf{pk}_i\}_{i \in [t]}$. The header contains the signature $\sigma = \text{Sign}(\mathbf{m}, \mathbf{sk}_{\text{agg}})$.

Light client verification \mathbf{V} receives, for $i \in [T - 1]$

1. A message $\mathbf{m}_i = (S_{i,1}, S_{i,2})$.
2. A string $x_i \in \{0, 1\}^t$
3. A signature σ_i .

\mathbf{V} starts with the validator set C_1 which we assume is hard coded in the genesis block and agreed upon.

For each $i \in \{2, \dots, T\}$ \mathbf{V}

1. Checks that x_i has at least $2/3 \cdot t$ set bits, and sets $D_i \subset [t]$ to be the indices of the set bits of x_i .
2. Computes $\mathbf{pk} := \sum_{j \in D_i} \mathbf{pk}_j$.
3. computes $C_{i+1} = (C_i \setminus S_{i,1}) \cup S_{i,2}$.

3 Hash to group

The BLS signature scheme uses $H : \{0, 1\}^* \rightarrow \mathbb{G}_2$ as a hash function that outputs random elements in \mathbb{G}_2 . It is modeled as a random oracle.

It is instantiated using a composition of a Pedersen hash defined over the curve $E_{\text{Ed}/\text{CP}}$ from [BCG⁺18], and a few Blake2 hashes.

Pedersen hash The Pedersen hash takes input strings of arbitrary length and outputs elements in the group $G_{E_{\text{Ed}/\text{CP}}}$ of $E_{\text{Ed}/\text{CP}}$ - **PedersenHash** : $\{0, 1\}^* \rightarrow G_{E_{\text{Ed}/\text{CP}}}$. Each group element is defined by $(x, y) \in \mathbb{F}_p^2$, where p has a bit size of 377, and therefore a byte size of 48. We then define **ULPSerialize** $(x, y) : \mathbb{F}_p^2 \rightarrow \{0, 1\}^{392}$ as **ULPSerialize** $(x, y) := x$. Finally, we define **ULPPedersenHash** : $\{0, 1\}^* \rightarrow \{0, 1\}^{392}$ as **ULPSerialize** \circ **PedersenHash**.

Hash to field To hash into the group \mathbb{G}_2 , where each element is defined by $(x, y) \in \mathbb{F}_{p^2}^2$, we first have to hash into the field \mathbb{F}_{p^2} . We invoke **Blake2s** multiple times to get enough random-looking bits to generate $x = (x_0, x_1)$ and take each inner element modulo p . We generate extra bits reduce modulo bias. Then, we see if x is a valid x on the curve, by trying to find a matching y .

Specifically, given a function we define **ULFieldHash** $(m) : \{0, 1\}^* \rightarrow \mathbb{F}_p$ as follows:

First, we calculate 1024 random-looking bits using:

$$\begin{aligned} & \mathbf{Blake2s}(0x00000000||m) \parallel \mathbf{Blake2s}(0x00000001||m) \parallel \\ & \mathbf{Blake2s}(0x00000002||m) \parallel \mathbf{Blake2s}(0x00000003||m) \end{aligned}$$

We then divide it into two 512 bits for x_0 and x_1 , and for each we parse the bits as a little-endian integer and take it $\bmod p$.

Hash to \mathbb{G}_2 We use the try-and-increment method - we initialize a counter, hash the counter together with the message as a possible x value, attempt to find a matching y and if we succeed, we multiply by the cofactor.

Algorithm 1: Try-and-increment hashing to the group

- Input** : m , a message
- 1 $hm := \mathbf{ULPedersenHash}(m)$
 - 2 Initialize $c := 0$
 - 3 Serialize c as a little-endian 32-bit unsigned integer, and store in cb
 - 4 $x := \mathbf{ULFieldHash}(c||hm)$
 - 5 Attempt finding a matching y by calculating $\sqrt{x^2 + 1}$. If not successful, increment c and go back to step 2.
 - 6 If successful, take the larger y , and return $\mathbf{cofactor} \cdot (x, y)$. .
-

References

- [BCG⁺18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. *IACR ePrint*, 962, 2018.