Belenios specification

Stéphane Glondu

Version 0.2

Contents

1 Introduction

This document is a specification of the voting protocol implemented in Belenios v0.2. More discussion, theoretical explanations and bibliographical references can be found in a technical report available online.[1]

The Belenios protocol is very similar to Helios (with a signature added to ballots and different zero-knowledge proofs) and Helios-C (with the distributed key generation of trustees of Helios, without threshold support).

[1] `http://eprint.iacr.org/2013/177`

The cryptography involved in Belenios needs a cyclic group $\mathbb{G}$ where discrete logarithms are hard to compute. We will denote by g a generator and q its order. We use a multiplicative notation for the group operation. For practical purposes, we use a multiplicative subgroup of $\mathbb{F}_p^*$ (hence, all exponentiations are implicitly done modulo p). We suppose the group parameters are agreed on beforehand. Default group parameters are given as examples in section 5.

2 Parties

- S: voting server
- A: server administrator
- C: credential authority
- $T_1, \dots, T_m$: trustees
- $V_1, \dots, V_n$: voters

3 Processes

3.1 Election setup

1. A generates a fresh `uuid` u and sends it to C
2. C generates credentials $c_1, \dots, c_n$ and computes $L = \mathsf{shuffle}(\mathsf{public}(c_1), \dots, \mathsf{public}(c_n))$
3. for $j \in [1 \dots n]$, C sends c_j to V_j
4. C forgets $c_1, \dots, c_n$
5. C forgets the mapping between j and $\mathsf{public}(c_j)$ if credential recovery is not needed
6. C sends L to A
7. for $z \in [1 \dots m]$,
   (a) T_z generates a `trustee_public_key` k_z and sends it to A
   (b) A checks k_z
8. A combines all the trustee public keys into the election public key y
9. A creates the `election` E
10. A loads E and L into S and starts it

3.2 Vote

1. V gets E
2. V creates a `ballot` b and submits it to S
3. S validates b and publishes it

3.3 Credential recovery

1. V contacts C
2. C looks up V's public credential $\mathsf{public}(c_i)$ and generates a new credential c_i'
3. C sends c_i' to V and forgets it
4. C sends $\mathsf{public}(c_i)$ and $\mathsf{public}(c_i')$ to A
5. A checks that $\mathsf{public}(c_i)$ has not been used and replaces it by $\mathsf{public}(c_i')$ in L

3.4 Tally

1. A stops S and computes the `encrypted_tally` Π
2. for $z \in [1 \dots m]$,
   (a) A sends Π to T_z
   (b) T_z generates a `partial_decryption` δ_z and sends it to A
   (c) A verifies δ_z
3. A combines all the partial decryptions, computes and publishes the election `result`

4 Messages

4.1 Conventions

Structured data is encoded in JSON (RFC 4627). There is no specific requirement on the formatting and order of fields, but care must be taken when hashes are computed. We use the notation $\mathsf{field}(o)$ to access the field field of o.

4.2 Basic types

- `string`: JSON string
- `uuid`: UUID (see RFC 4122), encoded as a JSON string
- $\mathbb{I}$: small integer, encoded as a JSON number
- $\mathbb{B}$: boolean, encoded as a JSON boolean
- $\mathbb{N}$, $\mathbb{Z}_q$, $\mathbb{G}$: big integer, written in base 10 and encoded as a JSON string

4.3 Common structures

$$\texttt{proof} = \left\{ \begin{array}{rcl} \mathsf{challenge} & : & \mathbb{Z}_q \\ \mathsf{response} & : & \mathbb{Z}_q \end{array} \right\} \qquad \texttt{ciphertext} = \left\{ \begin{array}{rcl} \mathsf{alpha} & : & \mathbb{G} \\ \mathsf{beta} & : & \mathbb{G} \end{array} \right\}$$

4.4 Trustee keys

$$\texttt{public\_key} = \mathbb{G} \qquad \texttt{private\_key} = \mathbb{Z}_q$$

$$\texttt{trustee\_public\_key} = \left\{ \begin{array}{rcl} \texttt{pok} & : & \texttt{proof} \\ \texttt{public\_key} & : & \texttt{public\_key} \end{array} \right\}$$

A private key is a random number x modulo q. The corresponding $\mathsf{public\_key}$ is $X = g^x$. A `trustee_public_key` is a bundle of this public key with a `proof` of knowledge computed as follows:

1. pick a random $w \in \mathbb{Z}_q$
2. compute $A = g^w$
3. $\mathsf{challenge} = \mathcal{H}_{\mathsf{pok}}(X, A) \mod q$
4. $\mathsf{response} = w + x \times \mathsf{challenge} \mod q$

where $\mathcal{H}_{\mathsf{pok}}$ is computed as follows:

$$\mathcal{H}_{\mathsf{pok}}(X, A) = \mathsf{SHA256}(\texttt{pok|}X\texttt{|}A)$$

where `pok` and the vertical bars are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number. The proof is verified as follows:

1. compute $A = g^{\mathsf{response}}/y^{\mathsf{challenge}}$
2. check that $\mathsf{challenge} = \mathcal{H}_{\mathsf{pok}}(\mathsf{public\_key}, A) \mod q$

4.5 Credentials

A secret *credential* c is a 15-character string, where characters are taken from the set:

`123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

The first 14 characters are random, and the last one is a checksum to detect typing errors. To compute the checksum, each character is interpreted as a base 58 digit: `1` is 0, `2` is 1, ..., `z` is 57. The first 14 characters are interpreted as a big-endian number c_1 The checksum is $53 - c_1 \mod 53$.

From this string, a secret exponent $s = \texttt{secret}(c)$ is derived by using PBKDF2 (RFC 2898) with:

- c as password;
- HMAC-SHA256 (RFC 2104, FIPS PUB 180-2) as pseudorandom function;
- the `uuid` (interpreted as a 16-byte array) of the election as salt;
- 1000 iterations

and an output size of 1 block, which is interpreted as a big-endian 256-bit number and then reduced modulo q to form s. From this secret exponent, a public key $\mathsf{public}(c) = g^s$ is computed.

4.6 Election

$$\texttt{wrapped\_pk} = \left\{ \begin{array}{ccc} \mathsf{g} & : & \mathbb{G} \\ \mathsf{p} & : & \mathbb{N} \\ \mathsf{q} & : & \mathbb{N} \\ \mathsf{y} & : & \mathbb{G} \end{array} \right\}$$

The election public key, which is denoted by y thoughout this document, is computed by multiplying all the public keys of the trustees, and bundled with the group parameters in a `wrapped_pk` structure.

$$\texttt{question} = \left\{ \begin{array}{rcl} \mathsf{answers} & : & \texttt{string}^* \\ ?\mathsf{blank} & : & \mathbb{B} \\ \mathsf{min} & : & \mathbb{I} \\ \mathsf{max} & : & \mathbb{I} \\ \mathsf{question} & : & \texttt{string} \end{array} \right\} \quad \texttt{election} = \left\{ \begin{array}{rcl} \mathsf{description} & : & \texttt{string} \\ \mathsf{name} & : & \texttt{string} \\ \mathsf{public\_key} & : & \texttt{wrapped\_pk} \\ \mathsf{questions} & : & \texttt{question}^* \\ \mathsf{uuid} & : & \texttt{uuid} \end{array} \right\}$$

The blank field of `question` is optional. When present and true, the voter can vote blank for this question. In a blank vote, all answers are set to 0 regardless of the values of min and max (min doesn't need to be 0).

During an election, the following data needs to be public in order to verify the setup phase and to validate ballots:

- the `election` structure described above;
- all the `trustee_public_key`s that were generated during the setup phase;
- the set L of public credentials.

4.7 Encrypted answers

$$\texttt{answer} = \left\{ \begin{array}{rcl} \mathsf{choices} & : & \texttt{ciphertext}^* \\ \mathsf{individual\_proofs} & : & \texttt{iproof}^* \\ \mathsf{overall\_proof} & : & \texttt{iproof} \\ ?\mathsf{blank\_proof} & : & \texttt{proof}^2 \end{array} \right\}$$

An answer to a `question` is the vector choices of encrypted weights given to each answer. When blank is false (or absent), a blank vote is not allowed and this vector has the same length as answers; otherwise, a blank vote is allowed and this vector has an additionnal leading weight corresponding to whether the vote is blank or not. Each weight comes with a proof (in individual_proofs, same length as choices) that it is 0 or 1. The whole answer also comes with additional proofs that weights respect constraints.

More concretely, each weight $m \in [0 \dots 1]$ is encrypted into a `ciphertext` as follows:

1. pick a random $r \in \mathbb{Z}_q$
2. $\mathsf{alpha} = g^r$
3. $\mathsf{beta} = y^r g^m$

where y is the election public key.

To compute the proofs, the voter needs a credential c. Let $s = \texttt{secret}(c)$, and $S = g^s$ written in base 10. The individual proof that $m \in [0 \dots 1]$ is computed by running $\mathsf{iprove}(S, r, m, 0, 1)$ (see section 4.8).

When a blank vote is not allowed, overall_proof proves that $M \in [\mathsf{min} \dots \mathsf{max}]$ and is computed by running $\mathsf{iprove}(S, R, M - \mathsf{min}, \mathsf{min}, \dots, \mathsf{max})$ where R is the sum of the r used in ciphertexts, and M the sum of the m. There is no blank_proof.

When a blank vote is allowed, and there are n choices, the answer is modeled as a vector $(m_0, m_1, \ldots, m_n)$, when m_0 is whether this is a blank vote or not, and m_i (for $i > 0$) is whether choice i has been selected. Each m_i is encrypted and proven equal to 0 or 1 as above. Let $m_\Sigma = m_1 + \cdots + m_n$. The additionnal proofs are as follows:

- $\mathsf{blank\_proof}$ proves that $m_0 = 0 \vee m_\Sigma = 0$;
- $\mathsf{overall\_proof}$ proves that $m_0 = 1 \vee m_\Sigma \in [\mathsf{min} \ldots \mathsf{max}]$.

They are computed as described in section 4.9.

4.8 Proofs of interval membership

$$\texttt{iproof} = \texttt{proof}^*$$

Given a pair (α, β) of group elements, one can prove that it has the form $(g^r, y^r g^{M_i})$ with $M_i \in [M_0, \ldots, M_k]$ by creating a sequence of $\texttt{proofs}$ $\pi_0, \ldots, \pi_k$ with the following procedure, parameterised by a group element S:

1. for $j \neq i$:
   (a) create π_j with a random $\mathsf{challenge}$ and a random $\mathsf{response}$
   (b) compute
   $$A_j = \frac{g^{\mathsf{response}}}{\alpha^{\mathsf{challenge}}} \quad \text{and} \quad B_j = \frac{y^{\mathsf{response}}}{(\beta/g^{M_j})^{\mathsf{challenge}}}$$
2. π_i is created as follows:
   (a) pick a random $w \in \mathbb{Z}_q$
   (b) compute $A_i = g^w$ and $B_i = y^w$
   (c) $\mathsf{challenge}(\pi_i) = \mathcal{H}_{\mathsf{iprove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) - \sum_{j \neq i} \mathsf{challenge}(\pi_j) \mod q$
   (d) $\mathsf{response}(\pi_i) = w + r \times \mathsf{challenge}(\pi_i) \mod q$

In the above, $\mathcal{H}_{\mathsf{iprove}}$ is computed as follows:

$$\mathcal{H}_{\mathsf{iprove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) = \mathsf{SHA256}(\texttt{prove|}S\texttt{|}\alpha\texttt{,}\beta\texttt{|}A_0\texttt{,}B_0\texttt{,}\ldots\texttt{,}A_k\texttt{,}B_k) \mod q$$

where $\texttt{prove}$, the vertical bars and the commas are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number. We will denote the whole procedure by $\mathsf{iprove}(S, r, i, M_0, \ldots, M_k)$.

The proof is verified as follows:

1. for $j \in [0 \ldots k]$, compute
   $$A_j = \frac{g^{\mathsf{response}(\pi_j)}}{\alpha^{\mathsf{challenge}(\pi_j)}} \quad \text{and} \quad B_j = \frac{y^{\mathsf{response}(\pi_j)}}{(\beta/g^{M_j})^{\mathsf{challenge}(\pi_j)}}$$
2. check that
   $$\mathcal{H}_{\mathsf{iprove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) = \sum_{j \in [0 \ldots k]} \mathsf{challenge}(\pi_j) \mod q$$

4.9 Proofs of possibly-blank votes

In this section, we suppose:

$$(\alpha_0, \beta_0) = (g^{r_0}, y^{r_0} g^{m_0}) \quad \text{and} \quad (\alpha_\Sigma, \beta_\Sigma) = (g^{r_\Sigma}, y^{r_\Sigma} g^{m_\Sigma})$$

Note that α_Σ, β_Σ and r_Σ can be easily computed from the encryptions of $m_1, \ldots, m_n$ and their associated secrets.

Additionnally, let P be the string "g,y,α_0,β_0,α_Σ,β_Σ", where the commas are verbatim and the numbers are written in base 10. Let also $M_1, \ldots, M_k$ be the sequence $\mathsf{min}, \ldots, \mathsf{max}$ ($k = \mathsf{max} - \mathsf{min} + 1$).

4.9.1 Non-blank votes ($m_0 = 0$)

Computing blank_proof In $m_0 = 0 \vee m_\Sigma = 0$, the first case is true. The proof blank_proof of the whole statement is the couple of proofs (π_0, π_Σ) built as follows:

1. pick random $\mathsf{challenge}(\pi_\Sigma)$ and $\mathsf{response}(\pi_\Sigma)$ in $\mathbb{Z}_q$
2. compute $A_\Sigma = g^{\mathsf{response}(\pi_\Sigma)} \times \alpha_\Sigma^{\mathsf{challenge}(\pi_\Sigma)}$ and $B_\Sigma = y^{\mathsf{response}(\pi_\Sigma)} \times \beta_\Sigma^{\mathsf{challenge}(\pi_\Sigma)}$
3. pick a random w in $\mathbb{Z}_q$
4. compute $A_0 = g^w$ and $B_0 = y^w$
5. compute

$$\mathsf{challenge}(\pi_0) = \mathcal{H}_{\mathsf{bproof0}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) - \mathsf{challenge}(\pi_\Sigma) \mod q$$

6. compute $\mathsf{response}(\pi_0) = w - r_0 \times \mathsf{challenge}(\pi_0) \mod q$

In the above, $\mathcal{H}_{\mathsf{bproof0}}$ is computed as follows:

$$\mathcal{H}_{\mathsf{bproof0}}(\ldots) = \mathsf{SHA256}(\texttt{bproof0|}S\texttt{|}P\texttt{|}A_0\texttt{,}B_0\texttt{,}A_\Sigma\texttt{,}B_\Sigma) \mod q$$

where `bproof0`, the vertical bars and the commas are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number.

Computing overall_proof In $m_0 = 1 \vee m_\Sigma \in [M_1 \ldots M_k]$, the second case is true. Let i be such that $m_\Sigma = M_i$. The proof of the whole statement is a $(k+1)$-tuple $(\pi_0, \pi_1, \ldots, \pi_k)$ built as follows:

1. pick random $\mathsf{challenge}(\pi_0)$ and $\mathsf{response}(\pi_0)$ in $\mathbb{Z}_q$
2. compute $A_0 = g^{\mathsf{response}(\pi_0)} \times \alpha_0^{\mathsf{challenge}(\pi_0)}$ and $B_0 = y^{\mathsf{response}(\pi_0)} \times (\beta_0/g)^{\mathsf{challenge}(\pi_0)}$
3. for $j > 0$ and $j \neq i$:
   (a) create π_j with a random challenge and a random response in $\mathbb{Z}_q$
   (b) compute $A_j = g^{\mathsf{response}} \times \alpha_\Sigma^{\mathsf{challenge}}$ and $B_j = y^{\mathsf{response}} \times (\beta_\Sigma/g^{M_j})^{\mathsf{challenge}}$
4. pick a random $w \in \mathbb{Z}_q$
5. compute $A_i = g^w$ and $B_i = y^w$
6. compute

$$\mathsf{challenge}(\pi_i) = \mathcal{H}_{\mathsf{bproof1}}(S, P, A_0, B_0, \ldots, A_k, B_k) - \sum_{j \neq i} \mathsf{challenge}(\pi_j) \mod q$$

7. compute $\mathsf{response}(\pi_i) = w - r_\Sigma \times \mathsf{challenge}(\pi_i) \mod q$

In the above, $\mathcal{H}_{\mathsf{bproof1}}$ is computed as follows:

$$\mathcal{H}_{\mathsf{bproof1}}(\dots) = \mathsf{SHA256}(\texttt{bproof1|}S\texttt{|}P\texttt{|}A_0\texttt{,}B_0\texttt{,}\dots\texttt{,}A_k\texttt{,}B_k) \mod q$$

where `bproof1`, the vertical bars and the commas are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number.

4.9.2 Blank votes ($m_0 = 1$)

Computing blank_proof In $m_0 = 0 \vee m_\Sigma = 0$, the second case is true. The proof blank_proof of the whole statement is the couple of proofs (π_0, π_Σ) built as in section 4.9.1, but exchanging subscripts 0 and Σ everywhere except in the call to $\mathcal{H}_{\mathsf{bproof0}}$.

Computing overall_proof In $m_0 = 1 \vee m_\Sigma \in [M_1 \dots M_k]$, the first case is true. The proof of the whole statement is a $(k+1)$-tuple $(\pi_0, \pi_1, \dots, \pi_k)$ built as follows:

1. for $j > 0$:
   (a) create π_j with a random challenge and a random response in $\mathbb{Z}_q$
   (b) compute $A_j = g^{\mathsf{response}} \times \alpha_\Sigma^{\mathsf{challenge}}$ and $B_j = y^{\mathsf{response}} \times (\beta_\Sigma / g^{M_j})^{\mathsf{challenge}}$
2. pick a random $w \in \mathbb{Z}_q$
3. compute $A_0 = g^w$ and $B_0 = y^w$
4. compute
$$\mathsf{challenge}(\pi_0) = \mathcal{H}_{\mathsf{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) - \sum_{j>0} \mathsf{challenge}(\pi_j) \mod q$$
5. compute $\mathsf{response}(\pi_0) = w - r_0 \times \mathsf{challenge}(\pi_0) \mod q$

4.9.3 Verifying proofs

Verifying blank_proof A proof of $m_0 = 0 \vee m_\Sigma = 0$ is a couple of proofs (π_0, π_Σ) such that the following procedure passes:

1. compute $A_0 = g^{\mathsf{response}(\pi_0)} \times \alpha_0^{\mathsf{challenge}(\pi_0)}$ and $B_0 = y^{\mathsf{response}(\pi_0)} \times \beta_0^{\mathsf{challenge}(\pi_0)}$
2. compute $A_\Sigma = g^{\mathsf{response}(\pi_\Sigma)} \times \alpha_\Sigma^{\mathsf{challenge}(\pi_\Sigma)}$ and $B_\Sigma = y^{\mathsf{response}(\pi_\Sigma)} \times \beta_\Sigma^{\mathsf{challenge}(\pi_\Sigma)}$
3. check that
$$\mathcal{H}_{\mathsf{bproof0}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) = \mathsf{challenge}(\pi_0) + \mathsf{challenge}(\pi_\Sigma) \mod q$$

Verifying overall_proof A proof of $m_0 = 1 \vee m_\Sigma \in [M_1 \dots M_k]$ is a $(k+1)$-tuple $(\pi_0, \pi_1, \dots, \pi_k)$ such that the following procedure passes:

1. compute $A_0 = g^{\mathsf{response}(\pi_0)} \times \alpha_0^{\mathsf{challenge}(\pi_0)}$ and $B_0 = y^{\mathsf{response}(\pi_0)} \times (\beta_0 / g)^{\mathsf{challenge}(\pi_0)}$
2. for $j > 0$, compute
$$A_j = g^{\mathsf{response}(\pi_j)} \times \alpha_j^{\mathsf{challenge}(\pi_j)} \quad \text{and} \quad B_j = y^{\mathsf{response}(\pi_j)} \times (\beta_j / g^{M_j})^{\mathsf{challenge}(\pi_j)}$$
3. check that
$$\mathcal{H}_{\mathsf{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) = \sum_{j=0}^{k} \mathsf{challenge}(\pi_j) \mod q$$

4.10 Signatures

$$\texttt{signature} = \left\{ \begin{array}{rcl} \textsf{public\_key} & : & \texttt{public\_key} \\ \textsf{challenge} & : & \mathbb{Z}_q \\ \textsf{response} & : & \mathbb{Z}_q \end{array} \right\}$$

Each ballot contains a digital signature to avoid ballot stuffing. The signature needs a credential c and uses all the `ciphertexts` $\gamma_1, \ldots, \gamma_l$ that appear in the ballot (l is the sum of the lengths of choices). It is computed as follows:

1. compute $s = \texttt{secret}(c)$
2. pick a random $w \in \mathbb{Z}_q$
3. compute $A = g^w$
4. $\textsf{public\_key} = g^s$
5. $\textsf{challenge} = \mathcal{H}_{\textsf{signature}}(\textsf{public\_key}, A, \gamma_1, \ldots, \gamma_l) \mod q$
6. $\textsf{response} = w - s \times \textsf{challenge} \mod q$

In the above, $\mathcal{H}_{\textsf{signature}}$ is computed as follows:

$$\mathcal{H}_{\textsf{signature}}(S, A, \gamma_1, \ldots, \gamma_l) = \textsf{SHA256}(\texttt{sig|}S\texttt{|}A\texttt{|}\textsf{alpha}(\gamma_1)\texttt{,}\textsf{beta}(\gamma_1)\texttt{,}\ldots\texttt{,}\textsf{alpha}(\gamma_l)\texttt{,}\textsf{beta}(\gamma_l))$$

where `sig`, the vertical bars and commas are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number.

Signatures are verified as follows:

1. compute $A = g^{\textsf{response}} \times \textsf{public\_key}^{\textsf{challenge}}$
2. check that $\textsf{challenge} = \mathcal{H}_{\textsf{signature}}(\textsf{public\_key}, A, \gamma_1, \ldots, \gamma_l) \mod q$

4.11 Ballots

$$\texttt{ballot} = \left\{ \begin{array}{rcl} \textsf{answers} & : & \texttt{answer}^* \\ \textsf{election\_hash} & : & \texttt{string} \\ \textsf{election\_uuid} & : & \texttt{uuid} \\ \textsf{signature} & : & \texttt{signature} \end{array} \right\}$$

The so-called hash (or *fingerprint*) of the election is computed with the function $\mathcal{H}_{\textsf{JSON}}$:

$$\mathcal{H}_{\textsf{JSON}}(J) = \textsf{BASE64}(\textsf{SHA256}(J))$$

Where J is the serialization (done by the server) of the `election` structure.

The same hashing function is used on a serialization (done by the voting client) of the `ballot` structure to produce a so-called *smart ballot tracker*.

4.12 Tally

$$\texttt{encrypted\_tally} = \texttt{ciphertext}^{**}$$

The encrypted tally is the pointwise product of the ciphertexts of all accepted ballots:

$$\begin{array}{rcl} \textsf{alpha}(\texttt{encrypted\_tally}_{i,j}) & = & \prod \textsf{alpha}(\textsf{choices}(\textsf{answers}(\texttt{ballot})_i)_j) \\ \textsf{beta}(\texttt{encrypted\_tally}_{i,j}) & = & \prod \textsf{beta}(\textsf{choices}(\textsf{answers}(\texttt{ballot})_i)_j) \end{array}$$

$$\texttt{partial\_decryption} = \left\{ \begin{array}{rcl} \mathsf{decryption\_factors} & : & \mathbb{G}^{**} \\ \mathsf{decryption\_proofs} & : & \texttt{proof}^{**} \end{array} \right\}$$

From the encrypted tally, each trustee computes a partial decryption using the private key x (and the corresponding public key $X = g^x$) he generated during election setup. It consists of so-called decryption factors:

$$\mathsf{decryption\_factors}_{i,j} = \mathsf{alpha}(\texttt{encrypted\_tally}_{i,j})^x$$

and proofs that they were correctly computed. Each $\mathsf{decryption\_proofs}_{i,j}$ is computed as follows:

1. pick a random $w \in \mathbb{Z}_q$
2. compute $A = g^w$ and $B = \mathsf{alpha}(\texttt{encrypted\_tally}_{i,j})^w$
3. $\mathsf{challenge} = \mathcal{H}_{\mathsf{decrypt}}(X, A, B)$
4. $\mathsf{response} = w + x \times \mathsf{challenge} \mod q$

In the above, $\mathcal{H}_{\mathsf{decrypt}}$ is computed as follows:

$$\mathcal{H}_{\mathsf{decrypt}}(X, A, B) = \mathsf{SHA256}(\texttt{decrypt|}X\texttt{|}A\texttt{,}B) \mod q$$

where `decrypt`, the vertical bars and the comma are verbatim and numbers are written in base 10. The result is interpreted as a 256-bit big-endian number.

These proofs are verified using the `trustee_public_key` structure k that the trustee sent to the administrator during the election setup:

1. compute
$$A = \frac{g^{\mathsf{response}}}{\mathsf{public\_key}(k)^{\mathsf{challenge}}} \quad \text{and} \quad B = \frac{\mathsf{alpha}(\texttt{encrypted\_tally}_{i,j})^{\mathsf{response}}}{\mathsf{decryption\_factors}_{i,j}^{\mathsf{challenge}}}$$
2. check that $\mathcal{H}_{\mathsf{decrypt}}(\mathsf{public\_key}(k), A, B) = \mathsf{challenge}$

4.13 Election result

$$\texttt{result} = \left\{ \begin{array}{rcl} \mathsf{num\_tallied} & : & \mathbb{I} \\ \mathsf{encrypted\_tally} & : & \texttt{encrypted\_tally} \\ \mathsf{partial\_decryptions} & : & \texttt{partial\_decryption}^{*} \\ \mathsf{result} & : & \mathbb{I}^{**} \end{array} \right\}$$

The decryption factors are combined for each ciphertext to build synthetic ones:

$$F_{i,j} = \prod_{z \in [1 \dots m]} \mathsf{partial\_decryptions}_{z,i,j}$$

where m is the number of trustees. The result field of the `result` structure is then computed as follows:

$$\mathsf{result}_{i,j} = \log_g \left(\frac{\mathsf{beta}(\mathsf{encrypted\_tally}_{i,j})}{F_{i,j}} \right)$$

Here, the discrete logarithm can be easily computed because it is bounded by $\mathsf{num\_tallied}$.

After the election, the following data needs to be public in order to verify the tally:

- the `election` structure;
- all the `trustee_public_key`s that were generated during the setup phase;
- the set of public credentials;
- the set of ballots;
- the `result` structure described above.

5 Default group parameters

These parameters have been generated by the `fips.sage` script (available in Belenios sources), which is itself based on FIPS 186-4.

$$
\begin{aligned}
p \quad = \quad & \hfill 20694785691422546 \\
& 401013643657505008064922989295751104097100884787057374219242 \\
& 717401922237254497684338129066633138078958404960054389636289 \\
& 796393038773905722803605973749427671376777618898589872735865 \\
& 049081167099310535867780980030790491654063777173764198678527 \\
& 273474476341835600035698305193144284561701911000786737307333 \\
& 564123971732897913240474578834468260652327974647951137672658 \\
& 693582180046317922073668860052627186363386088796882120769432 \\
& 366149491002923444346373222145884100586421050242120365433561 \\
& 201320481118852408731077014151666200162313177169372189248078 \\
& 507711827842317498073276598828825169183103125680162072880719 \\
g \quad = \quad & \hfill 2402352677501852 \\
& 209227687703532399932712287657378364916510075318787663274146 \\
& 353219320285676155269678799694668298749389095083896573425601 \\
& 900601068477164491735474137283104610458681314511781646755400 \\
& 527402889846139864532661215055797097162016168270312886432456 \\
& 663834863635782106154918419982534315189740658186868651151358 \\
& 576410138882215396016043228843603930989333662772848406593138 \\
& 406010231675095763777982665103606822406635076697764025346253 \\
& 773085133173495194248967754052573659049492477631475991575198 \\
& 775177711481490920456600205478127054728238140972518639858334 \\
& 115700568353695553423781475582491896050296680037745308460627 \\
q \quad = \quad & \hfill 78571733251071885 \\
& 079927659812671450121821421258408794611510081919805623223441
\end{aligned}
$$

The additional output of the generation algorithm is:

$$
\begin{aligned}
\texttt{domain\_parameter\_seed} \quad = \quad & \hfill 478953892617249466 \\
& 166106476098847626563138168027 \\
& 716882488732447198349000396592 \\
& 020632875172724552145560167746 \\
\texttt{counter} \quad = \quad & \hfill 109
\end{aligned}
$$