Belenios specification

Stéphane Glondu

Version 1.6

Contents

1 Introduction

This document is a specification of the voting protocol implemented in Belenios v1.6. More discussion, theoretical explanations and bibliographical references can be found in a technical report available online.[1]

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

- $\mathcal{S}$: voting server
- $\mathcal{A}$: server administrator
- $\mathcal{C}$: credential authority
- $\mathcal{T}_1, \dots, \mathcal{T}_m$: trustees
- $\mathcal{V}_1, \dots, \mathcal{V}_n$: voters

3 Processes

3.1 Election setup

1. $\mathcal{A}$ generates a fresh `uuid` u and sends it to $\mathcal{C}$
2. $\mathcal{C}$ generates credentials $c_1, \dots, c_n$ and computes $L = \mathsf{shuffle}(\mathsf{public}(c_1), \dots, \mathsf{public}(c_n))$
3. for $j \in [1 \dots n]$, $\mathcal{C}$ sends c_j to $\mathcal{V}_j$
4. $\mathcal{C}$ forgets $c_1, \dots, c_n$
5. $\mathcal{C}$ forgets the mapping between j and $\mathsf{public}(c_j)$ if credential recovery is not needed
6. $\mathcal{C}$ sends L to $\mathcal{A}$
7. $\mathcal{A}$ and $\mathcal{T}_1, \dots, \mathcal{T}_m$ run a key establishment protocol (either 3.1.1 or 3.1.2)
8. $\mathcal{A}$ creates the `election` E
9. $\mathcal{A}$ loads E and L into $\mathcal{S}$ and starts it

3.1.1 Basic decryption support

To perform tally with this scheme, all trustees will need to compute a partial decryption.

1. for $z \in [1 \dots m]$,
   (a) $\mathcal{T}_z$ generates a `trustee_public_key` k_z and sends it to $\mathcal{A}$
   (b) $\mathcal{A}$ checks k_z
2. $\mathcal{A}$ combines all the trustee public keys into the election public key y

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

3.1.2 Threshold decryption support

To perform tally with this scheme, $t+1$ trustees will need to compute a partial decryption.

1. for $z \in [1 \dots m]$,
   (a) $\mathcal{T}_z$ generates a `cert` γ_z and sends it to $\mathcal{A}$
   (b) $\mathcal{A}$ checks γ_z
2. $\mathcal{A}$ assembles $\Gamma = \gamma_1, \dots, \gamma_n$
3. for $z \in [1 \dots m]$,
   (a) $\mathcal{A}$ sends Γ to $\mathcal{T}_z$ and $\mathcal{T}_z$ checks it
   (b) $\mathcal{T}_z$ generates a `polynomial` P_z and sends it to $\mathcal{A}$
   (c) $\mathcal{A}$ checks P_z
4. for $z \in [1 \dots m]$, $\mathcal{A}$ computes a `vinput` vi_z
5. for $z \in [1 \dots m]$,
   (a) $\mathcal{A}$ sends Γ to $\mathcal{T}_z$ and $\mathcal{T}_z$ checks it
   (b) $\mathcal{A}$ sends vi_z to $\mathcal{T}_z$ and $\mathcal{T}_z$ checks it
   (c) $\mathcal{T}_z$ computes a `voutput` vo_z and sends it to $\mathcal{A}$
   (d) $\mathcal{A}$ checks vo_z
6. $\mathcal{A}$ extracts encrypted decryption keys $K_1, \dots, K_m$ and threshold parameters

3.2 Vote

1. $\mathcal{V}$ gets E
2. $\mathcal{V}$ creates a `ballot` b and submits it to $\mathcal{S}$
3. $\mathcal{S}$ validates b and publishes it

3.3 Credential recovery

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

3.4 Tally

1. $\mathcal{A}$ stops $\mathcal{S}$ and computes the `encrypted_tally` Π
2. for $z \in [1 \dots m]$ (or, if in threshold mode, a subset of it of size at least $t+1$),
   (a) $\mathcal{A}$ sends Π (and K_z if in threshold mode) to $\mathcal{T}_z$
   (b) $\mathcal{T}_z$ generates a `partial_decryption` δ_z and sends it to $\mathcal{A}$
   (c) $\mathcal{A}$ verifies δ_z
3. $\mathcal{A}$ 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 (either as defined in RFC 4122, or a string of Base58 characters[2] of size at least 14), 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

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

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

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:

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

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 $\mathsf{challenge} = \mathcal{H}_{\mathsf{pok}}(\mathsf{public\_key}, A) \mod q$

[2]Base58 characters are: `123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

4.5 Messages specific to threshold decryption support

4.5.1 Public key infrastructure

Establishing a public key so that threshold decryption is supported requires private communications between trustees. To achieve this, Belenios uses a custom public key infrastructure. During the key establishment protocol, each trustee starts by generating a secret seed (at random), then derives from it encryption and decryption keys, as well as signing and verification keys. These four keys are then used to exchange messages between trustees by using $\mathcal{A}$ as a proxy.

The secret seed s is a 22-character string, where characters are taken from the set:

`123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

Deriving keys The (private) signing key sk is derived by computing the SHA256 of s prefixed by the string `sk|`. The corresponding (public) verification key is g^{sk}. The (private) decryption key dk is derived by computing the SHA256 of s prefixed by the string `dk|`. The corresponding (public) encryption key is g^{dk}.

Signing Signing takes a signing key sk and a $\mathsf{message}$ M (as a `string`), computes a $\mathsf{signature}$ and produces a `signed_msg`. For the signature, we use a (Schnorr-like) non-interactive zero-knowledge proof.

$$\texttt{signed\_msg} = \left\{ \begin{array}{ccc} \mathsf{message} & : & \texttt{string} \\ \mathsf{signature} & : & \texttt{proof} \end{array} \right\}$$

To compute the $\mathsf{signature}$,

1. pick a random $w \in \mathbb{Z}_q$
2. compute the commitment $A = g^w$
3. compute the $\mathsf{challenge}$ as $\mathsf{SHA256}(\texttt{sigmsg|}M\texttt{|}A)$, where A is written in base 10 and the result is interpreted as a 256-bit big-endian number
4. compute the $\mathsf{response}$ as $w - \mathsf{sk} \times \mathsf{challenge} \mod q$

To verify a $\mathsf{signature}$ using a verification key vk,

1. compute the commitment $A = g^{\mathsf{response}} \times \mathsf{vk}^{\mathsf{challenge}}$
2. check that $\mathsf{challenge} = \mathsf{SHA256}(\texttt{sigmsg|}M\texttt{|}A)$

Encrypting Encrypting takes an encryption key ek and a message M (as a `string`), computes an `encrypted_msg` and serializes it as a `string`. We use an El Gamal-like system.

$$\texttt{encrypted\_msg} = \left\{ \begin{array}{ccc} \mathsf{alpha} & : & \mathbb{G} \\ \mathsf{beta} & : & \mathbb{G} \\ \mathsf{data} & : & \texttt{string} \end{array} \right\}$$

To compute the `encrypted_msg`:

1. pick random $r, s \in \mathbb{Z}_q$
2. compute $\mathsf{alpha} = g^r$
3. compute $\mathsf{beta} = \mathsf{ek}^r \times g^s$

4. compute data as the hexadecimal encoding of the (symmetric) encryption of M using AES in CCM mode with $\mathsf{SHA256}(\texttt{key}|g^s)$ as the key and $\mathsf{SHA256}(\texttt{iv}|g^r)$ as the initialization vector (where numbers are written in base 10)

To decrypt an `encrypted_msg` using a decryption key dk:

1. compute the symmetric key as $\mathsf{SHA256}(\texttt{key}|\mathsf{beta}/(\mathsf{alpha}^{\mathsf{dk}}))$
2. compute the initialization vector as $\mathsf{SHA256}(\texttt{iv}|\mathsf{alpha})$
3. decrypt data

4.5.2 Certificates

A certificate is a `signed_msg` encapsulating a serialized `cert_keys` structure, itself filled with the public keys generated as described in section 4.5.1.

$$\texttt{cert} = \texttt{signed\_msg} \qquad \texttt{cert\_keys} = \left\{ \begin{array}{rcl} \mathsf{verification} & : & \mathbb{G} \\ \mathsf{encryption} & : & \mathbb{G} \end{array} \right\}$$

The message is signed with the signing key associated to verification.

4.5.3 Channels

A message is sent securely from sk (a signing key) to recipient (an encryption key) by encapsulating it in a `channel_msg`, serializing it as a `string`, signing it with sk and serializing the resulting `signed_msg` as a `string`, and finally encrypting it with recipient. The resulting `string` will be denoted by $\mathsf{send}(\mathsf{sk}, \mathsf{recipient}, \mathsf{message})$, and can be transmitted using a third-party (such as the election administrator).

$$\texttt{channel\_msg} = \left\{ \begin{array}{rcl} \mathsf{recipient} & : & \mathbb{G} \\ \mathsf{message} & : & \texttt{string} \end{array} \right\}$$

When decoding such a message, recipient must be checked.

4.5.4 Polynomials

Let $\Gamma = \gamma_1, \dots, \gamma_m$ be the certificates of all trustees. We will denote by vk_z (resp. ek_z) the verification key (resp. the encryption key) of γ_z. Each trustee must compute a `polynomial` structure in step 3 of the key establishment protocol.

$$\texttt{polynomial} = \left\{ \begin{array}{rcl} \mathsf{polynomial} & : & \texttt{string} \\ \mathsf{secrets} & : & \texttt{string}^* \\ \mathsf{coefexps} & : & \texttt{coefexps} \end{array} \right\}$$

Suppose $\mathcal{T}_i$ is the trustee who is computing. Therefore, $\mathcal{T}_i$ knows the signing key sk_i corresponding to vk_i and the decryption key dk_i corresponding to ek_i. $\mathcal{T}_i$ first checks that keys indeed match. Then $\mathcal{T}_i$ picks a random polynomial

$$f_i(x) = a_{i0} + a_{i1}x + \cdots + a_{it}x^t \in \mathbb{Z}_q[x]$$

and computes $A_{ik} = g^{a_{ik}}$ for $k = 0, \dots, t$ and $s_{ij} = f_i(j) \mod q$ for $j = 1, \dots, m$. $\mathcal{T}_i$ then fills the `polynomial` structure as follows:

- the polynomial field is $\mathsf{send}(\mathsf{sk}_i, \mathsf{ek}_i, M)$ where M is a serialized `raw_polynomial` structure

  $$\texttt{raw\_polynomial} = \left\{ \begin{array}{rcl} \mathsf{polynomial} & : & \mathbb{Z}_q^* \end{array} \right\}$$

  filled with $a_{i0}, \dots, a_{it}$

- the $\mathsf{secrets}$ field is $\mathsf{send}(\mathsf{sk}_i, \mathsf{ek}_1, M_{i1}), \ldots, \mathsf{send}(\mathsf{sk}_i, \mathsf{ek}_m, M_{im})$ where M_{ij} is a serialized `secret` structure

$$\texttt{secret} = \left\{ \begin{array}{rcl} \mathsf{secret} & : & \mathbb{Z}_q \end{array} \right\}$$

  filled with s_{ij}

- the $\mathsf{coefexps}$ field is a signed message containing a serialized `raw_coefexps` structure

$$\texttt{coefexps} = \texttt{signed\_msg} \qquad \texttt{raw\_coefexps} = \left\{ \begin{array}{rcl} \mathsf{coefexps} & : & \mathbb{G}^* \end{array} \right\}$$

  filled with $A_{i0}, \ldots, A_{it}$

4.5.5 Vinputs

Once we receive all the `polynomial` structures $P_1, \ldots, P_m$, we compute (during step 4) input data (called `vinput`) for a verification step performed later by the trustees. Step 4 can be seen as a routing step.

$$\texttt{vinput} = \left\{ \begin{array}{rcl} \mathsf{polynomial} & : & \texttt{string} \\ \mathsf{secrets} & : & \texttt{string}^* \\ \mathsf{coefexps} & : & \texttt{coefexps}^* \end{array} \right\}$$

Suppose we are computing the `vinput` structure vi_j for trustee $\mathcal{T}_j$. We fill it as follows:

- the $\mathsf{polynomial}$ field is the same as the one of P_j
- the secret field is $\mathsf{secret}(P_1)_j, \ldots, \mathsf{secret}(P_m)_j$
- the $\mathsf{coefexps}$ field is $\mathsf{coefexps}(P_1), \ldots, \mathsf{coefexps}(P_m)$

Note that the $\mathsf{coefexps}$ field is the same for all trustees.

In step 5, $\mathcal{T}_j$ checks consistency of vi_j by unpacking it and checking that, for $i = 1, \ldots, m$,

$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k}$$

4.5.6 Voutputs

In step 5 of the key establishment protocol, a trustee $\mathcal{T}_j$ receives Γ and vi_j, and produces a `voutput` vo_j.

$$\texttt{voutput} = \left\{ \begin{array}{rcl} \mathsf{private\_key} & : & \texttt{string} \\ \mathsf{public\_key} & : & \texttt{trustee\_public\_key} \end{array} \right\}$$

Trustee $\mathcal{T}_j$ fills vo_j as follows:

- $\mathsf{private\_key}$ is set to $\mathsf{send}(\mathsf{sk}_j, \mathsf{ek}_j, S_j)$, where S_j is $\mathcal{T}_j$'s (private) decryption key:

$$S_j = \sum_{i=1}^{m} s_{ij} \mod q$$

- $\mathsf{public\_key}$ is set to a `trustee_public_key` structure built using S_j as private key.

The administrator checks vo_j as follows:

- check that:

$$\mathsf{public\_key}(\mathsf{public\_key}(\mathsf{vo}_j)) = \prod_{i=1}^{m} \prod_{k=0}^{t} (A_{ik})^{j^k}$$

- check $\mathsf{pok}(\mathsf{public\_key}(\mathsf{vo}_j))$

4.5.7 Threshold parameters

The `threshold_parameters` structure embeds data that is published during the election.

$$\texttt{threshold\_parameters} = \left\{ \begin{array}{rcl} \textsf{threshold} & : & \mathbb{I} \\ \textsf{certs} & : & \texttt{cert}^* \\ \textsf{coefexps} & : & \texttt{coefexps}^* \\ \textsf{verification\_keys} & : & \texttt{trustee\_public\_key}^* \end{array} \right\}$$

The administrator fills it as follows:

- threshold is set to $t + 1$
- certs is set to $\Gamma = \gamma_1, \dots, \gamma_m$
- coefexps is set to the same value as the coefexps field of `vinputs`
- verification_keys is set to $\textsf{public\_key}(\textsf{vo}_1), \dots, \textsf{public\_key}(\textsf{vo}_m)$

4.6 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` (either interpreted as a 16-byte array in the RFC 4122 case, or directly itself in the Base58 case) 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 $\textsf{public}(c) = g^s$ is computed.

4.7 Election

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

The election public key, which is denoted by y thoughout this document, is computed during the setup phase, and bundled with the group parameters in a `wrapped_pk` structure.

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

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_keys, or the threshold_parameters, that were generated during the setup phase;
- the set L of public credentials.

4.8 Encrypted answers

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

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 $m \in [0 \ldots 1]$ is encrypted (in an El Gamal-like fashion) into a ciphertext as follows:

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

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 $m \in [0 \ldots 1]$ is computed by running $\textsf{iprove}(S, r, m, 0, 1)$ (see section 4.9).

When a blank vote is not allowed, overall_proof proves that $M \in [\textsf{min} \ldots \textsf{max}]$ and is computed by running $\textsf{iprove}(S, R, M - \textsf{min}, \textsf{min}, \ldots, \textsf{max})$ 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:

- blank_proof proves that $m_0 = 0 \vee m_\Sigma = 0$;
- overall_proof proves that $m_0 = 1 \vee m_\Sigma \in [\textsf{min} \ldots \textsf{max}]$.

They are computed as described in section 4.10.

4.9 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 $M_i \in [M_0, \dots, M_k]$ by creating a sequence of `proofs` $\pi_0, \dots, \pi_k$ 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) $\mathsf{challenge}(\pi_i) = \mathcal{H}_{\mathsf{iprove}}(S, \alpha, \beta, A_0, B_0, \dots, A_k, B_k) - \sum_{j \neq i} \mathsf{challenge}(\pi_j) \mod q$
   (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:

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

where `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 $\mathsf{iprove}(S, r, i, M_0, \dots, M_k)$.

The proof is verified as follows:

1. for $j \in [0 \dots k]$, 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
   $$\mathcal{H}_{\mathsf{iprove}}(S, \alpha, \beta, A_0, B_0, \dots, A_k, B_k) = \sum_{j \in [0 \dots k]} \mathsf{challenge}(\pi_j) \mod q$$

4.10 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 $m_1, \dots, m_n$ and their associated secrets.

Additionnally, let P be the string "$g\texttt{,}y\texttt{,}\alpha_0\texttt{,}\beta_0\texttt{,}\alpha_\Sigma\texttt{,}\beta_\Sigma$", where the commas are verbatim and the numbers are written in base 10. Let also $M_1, \dots, M_k$ be the sequence $\mathsf{min}, \dots, \mathsf{max}$ ($k = \mathsf{max} - \mathsf{min} + 1$).

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

Computing blank_proof In $m_0 = 0 \lor m_\Sigma = 0$, 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:

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

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 $m_0 = 1 \lor m_\Sigma \in [M_1 \dots M_k]$, 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 $(\pi_0, \pi_1, \dots, \pi_k)$ 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 $\mathsf{challenge}$ and a random $\mathsf{response}$ 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
$$\mathsf{challenge}(\pi_i) = \mathcal{H}_{\mathsf{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) - \sum_{j \neq i} \mathsf{challenge}(\pi_j) \mod q$$
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:

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

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.

4.10.2 Blank votes ($m_0 = 1$)

Computing $\mathsf{blank\_proof}$ In $m_0 = 0 \vee m_\Sigma = 0$, the second case is true. The proof $\mathsf{blank\_proof}$ of the whole statement is the couple of proofs (π_0, π_Σ) built as in section 4.10.1, but exchanging subscripts 0 and Σ everywhere except in the call to $\mathcal{H}_{\mathsf{bproof0}}$.

Computing $\mathsf{overall\_proof}$ 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 $\mathsf{challenge}$ and a random $\mathsf{response}$ 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 $\mathsf{response}(\pi_0) = w - r_0 \times \mathsf{challenge}(\pi_0) \mod q$

4.10.3 Verifying proofs

Verifying $\mathsf{blank\_proof}$ 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 $\mathsf{overall\_proof}$ 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

$$\mathcal{H}_{\mathsf{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) = \sum_{j=0}^{k} \mathsf{challenge}(\pi_j) \mod q$$

4.11 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 (Schnorr-like) 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 $\textsf{challenge} = \mathcal{H}_{\textsf{signature}}(\textsf{public\_key}, A, \gamma_1, \ldots, \gamma_l) \mod q$

4.12 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*.

4.13 Tally

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

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

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

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

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:

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

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

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

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

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

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

$$A = \frac{g^{\textsf{response}}}{\textsf{public\_key}(k)^{\textsf{challenge}}} \quad \text{and} \quad B = \frac{\textsf{alpha}(\texttt{encrypted\_tally}_{i,j})^{\textsf{response}}}{\textsf{decryption\_factors}_{i,j}^{\textsf{challenge}}}$$

2. check that $\mathcal{H}_{\textsf{decrypt}}(\textsf{public\_key}(k), A, B) = \textsf{challenge}$

4.14 Election result

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

The decryption factors are combined for each ciphertext to build synthetic ones $F_{i,j}$. With basic decryption support:

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

where m is the number of trustees. With threshold decryption support:

$$F_{i,j} = \prod_{z \in \mathcal{I}} (\textsf{partial\_decryptions}_{z,i,j})^{\lambda_z^{\mathcal{I}}}$$

where $\mathcal{I} = \{z_1, \dots, z_{t+1}\}$ is the set of indexes of supplied partial decryptions, and $\lambda_z^{\mathcal{I}}$ are the Lagrange coefficients:

$$\lambda_z^{\mathcal{I}} = \prod_{k \in \mathcal{I} \setminus \{z\}} \frac{k}{k - z} \mod q$$

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 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_keys`, or the `threshold_parameters`, 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.

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

The additional output of the generation algorithm is:

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