# Belenios specification

Stéphane Glondu

Version 1.11

## Contents

1. **Introduction**  
2. **Parties**  
3. **Processes**  
   3.1 Election setup  
   3.1.1 Basic decryption support  
   3.1.2 Threshold decryption support  
   3.2 Vote  
   3.3 Credential recovery  
   3.4 Tally  
   3.5 Audit  
      3.5.1 During the voting phase  
      3.5.2 After the tally  
4. **Messages**  
   4.1 Conventions  
   4.2 Basic types  
   4.3 Common structures  
   4.4 Verification keys  
   4.5 Messages specific to threshold decryption support  
      4.5.1 Public key infrastructure  
      4.5.2 Certificates  
      4.5.3 Channels  
      4.5.4 Polynomials  
      4.5.5 Vinputs  
      4.5.6 Voutputs  
      4.5.7 Threshold parameters  
   4.6 Trustees  
   4.7 Credentials  
   4.8 Questions  
      4.8.1 Homomorphic questions  
      4.8.2 Non-homomorphic questions  
   4.9 Elections  
   4.10 Encrypted answers  
      4.10.1 Homomorphic answers  
      4.10.2 Non-homomorphic answers  
   4.11 Proofs of interval membership  
   4.12 Proofs of possibly-blank votes  
      4.12.1 Non-blank votes ($m_0 = 0$)  
      4.12.2 Blank votes ($m_0 = 1$)
1 Introduction

References. This document is a specification of the voting protocol implemented in Belenios 1.11. A high level description of Belenios and some statistics about its usage can be found [6]. A security proof of the protocol for ballot privacy and verifiability is presented in [3]. The proof has been conducted with the tool EasyCrypt. It focuses on the protocol aspects and assumes security of the cryptographic primitives. The cryptographic primitives have been introduced in various places and their security proofs are spread across several references.

- The threshold decryption scheme is based on a “folklore” scheme: Pedersen’s [9] Distributed Key Generation (DKG) that has several variations. The variant considered in Belenios is described in [4] and proved in [4, 2].
- Ballots are composed of an ElGamal encryption of the votes and a zero-knowledge proof of well-formedness, as for the Helios protocol [1]. Compared to Helios, we support blank votes, which required to adapt the zero-knowledge proofs, as specified and proved in [7]. Additionally, ballots are signed to avoid ballot stuffing, as introduced in [5] and also described in [6].
- During the tally phase, Belenios supports two modes. Ballots are either combined homomorphically or shuffled and randomized, using mixnets. The mixnet algorithms are taken from the CHVote specification [8].

Types of supported elections. Belenios supports two main types of questions. In the homomorphic case, voters can select between $k_1$ and $k_2$ candidates out of $k$ candidates. This case is called homomorphic because the result of the election for such questions is the number of votes received for each candidate. No more information is leaked. In the non-homomorphic case, voters can give a number to each candidate. This can be used to rank candidates or grade them. Then the (raw) result of the election is simply the list of votes, as emitted by the voters, in a random order, to preserve privacy. Any counting method can then be applied (e.g. Condorcet, STV, or majority judgement) although Belenios does not offer support for this. The non-homomorphic case therefore offers much more flexibility, at the cost of extra steps during the tally (in order to securely shuffle the ballots). Belenios supports both types of questions and an election can even mix homomorphic and non-homomorphic questions.

Group parameters. The cryptography involved in Belenios needs a cyclic group $G$ 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

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

The voting server maintains the public data $D$ that consists of:

- the election data $E$
- the list $PK$ of public keys of the trustees
- the list $L$ of public credentials
- the list $B$ of accepted ballots
- the result of the election $\text{result}$ (once the election is tallied)

3 Processes

3.1 Election setup

1. $A$ generates a fresh $\text{uuid}$ $u$ and sends it to $C$
2. $C$ generates credentials $c_1, \ldots, c_n$ and computes $L = \text{shuffle}(\text{public}(c_1), \ldots, \text{public}(c_n))$
3. for $j \in [1 \ldots n]$, $C$ sends $c_j$ to $V_j$
4. (optional) $C$ forgets $c_1, \ldots, c_n$
5. $C$ sends $L$ to $A$
6. $A$ and $T_1, \ldots, T_m$ run a key establishment protocol (either 3.1.1 or 4.5)
7. $A$ creates the election $E$
8. $A$ loads $E$ and $L$ into $S$ and starts it
9. $C$ checks that the list of public credentials $L$ is exactly the one that appears on the election data of the election of $\text{uuid}$ $u$.

Step 4 is optional. It offers a better protection against ballot stuffing in case $C$ unintentionally leaks private credentials.

3.1.1 Basic decryption support

The trustees jointly compute the public election key. They will all need to contribute to the tally.

1. for $z \in [1 \ldots m]$,
   (a) $T_z$ generates a $\text{trustee_public_key}$ $k_z$ and sends it to $A$
   (b) $A$ checks $k_z$
2. $A$ combines all the trustee public keys into the election public key $y$:
   $$ y = \prod_{z \in [1 \ldots m]} \text{public_key}(k_z) $$
3. for $z \in [1 \ldots m]$, $T_z$ checks that $k_z$ appears in the set of public keys $PK$ of the election of $\text{uuid}$ $u$ (the id of the election should be publicly known).
### 3.1.2 Threshold decryption support

The trustees jointly compute the public election key such that only a subset of $t + 1$ of them will be needed to compute the tally.

1. for $z \in [1 \ldots m]$,
   (a) $T_z$ generates a $\text{cert} \gamma_z$ and sends it to $A$
   (b) $A$ checks $\gamma_z$

2. $A$ assembles $\Gamma = \gamma_1, \ldots, \gamma_m$

3. for $z \in [1 \ldots m]$,
   (a) $A$ sends $\Gamma$ to $T_z$ and $T_z$ checks it
   (b) $T_z$ generates a $\text{polynomial} P_z$ and sends it to $A$
   (c) $A$ checks $P_z$

4. for $z \in [1 \ldots m]$, $A$ computes a $\text{vinput} vi_z$

5. for $z \in [1 \ldots m]$,
   (a) $A$ sends $\Gamma$ to $T_z$ and $T_z$ checks it
   (b) $A$ sends $vi_z$ to $T_z$ and $T_z$ checks it
   (c) $T_z$ computes a $\text{voutput} vo_z$ and sends it to $A$
   (d) $A$ checks $vo_z$

6. $A$ extracts encrypted decryption keys $K_1, \ldots, K_m$ and threshold parameters

7. $A$ computes the election public key $y$ as specified in section 4.5.4

8. for $z \in [1 \ldots m]$, $T_z$ checks that $\gamma_z$ appears in the set of public keys $PK$ of the election of $\text{uuid} u$ (the id of the election should be publicly known).

### 3.2 Vote

1. $V$ gets $E$

2. $V$ creates a $\text{ballot} b$ and submits it to $S$

3. $S$ validates $b$ and adds it to $B$

4. at any time (even after tally), $V$ may check that $b$ appears in the list of accepted ballots $B$

### 3.3 Credential recovery

If $C$ has forgotten the private credentials of the voter (optional step 4 of the setup) then credentials cannot be recovered.

If $C$ has the list of private credentials (associated to the voters), credentials can be recovered:

1. $V_i$ contacts $C$

2. $C$ looks up $V_i$’s private credential $c_i$

3. $C$ sends $c_i$
3.4 Tally

1. \(\mathcal{A}\) stops \(\mathcal{S}\) and computes the initial encrypted tally \(\Pi_0\).

2. \(\mathcal{A}\) extracts the non-homomorphic ciphertexts from the encrypted tally (see section 4.16):
\[
\hat{\Pi}_0 = \text{nh_ciphertexts}(\Pi_0)
\]

3. if the election contains a non-homomorphic part, that is, if \(\hat{\Pi}_0 \neq []\), then for \(z \in [1 \ldots m]\):
   (a) \(\mathcal{A}\) sends \(\hat{\Pi}_{z-1}\) to \(\mathcal{T}_z\)
   (b) \(\mathcal{T}_z\) runs the shuffle algorithm, producing a \textit{shuffle} \(\sigma_z\) and sends it to \(\mathcal{A}\)
   (c) \(\mathcal{A}\) verifies \(\sigma_z\) and extracts \(\hat{\Pi}_z = \text{ciphertexts}(\sigma_z)\)

4. \(\mathcal{A}\) merges shuffled non-homomorphic ciphertexts with homomorphic ciphertexts, i.e. builds \(\Pi\) such that:
\[
\hat{\Pi}_m = \text{nh_ciphertexts}(\Pi)
\]

5. for \(z \in [1 \ldots m]\) (or, if in threshold mode, a subset of it of size at least \(t + 1\)),
   (a) \(\mathcal{A}\) sends \(\Pi\) (and \(K_z\) if in threshold mode) to \(\mathcal{T}_z\)
   (b) \(\mathcal{T}_z\) generates a \textit{partial decryption} \(\delta_z\) and sends it to \(\mathcal{A}\)
   (c) \(\mathcal{A}\) verifies \(\delta_z\)

6. \(\mathcal{A}\) combines all the partial decryptions, computes and publishes the election \textit{result}

7. \(\mathcal{T}_z\) checks that \(\delta_z\) and \(\sigma_z\) appears in \textit{result}

3.5 Audit

Belenios can be publicly audited: anyone having access to the (public) election data can check that the ballots are well formed and that the result corresponds to the ballots. Ideally, the list of ballots should also be monitored during the voting phase, to guarantee that no ballot disappears.

3.5.1 During the voting phase

At any time, an auditor can retrieve the public board and check its consistency. She should always record at least the last audited board. Then:

1. she retrieves the election data \(D = (E, PK, L, B, r)\) where \(B\) is the list of ballots:
   - she records \(D\);
   - for \(b \in B\), she checks that the proofs of \(b\) are valid and that the signature of \(b\) is valid and corresponds to one of the keys in \(L\);
   - she checks that any two ballots in \(B\) correspond to distinct keys (of \(L\));

2. she retrieves the previously recorded election data \(D' = (E', PK', L', B', r')\) (if it exists):
   - for \(b \in B'\), she checks that
     - \(b \in B\)
     - or \(\exists b' \in B\) such that \(b\) and \(b'\) correspond to the same key in \(L\). This corresponds to the case where a voter has revoted;
   - she checks that all the other data is unchanged: \(E = E', PK = PK', L = L', \) and \(r = r'\) (actually the result is empty at this step).

There is no tool support on the web interface for these checks, instead the command line tool \texttt{verify-diff} can be used.
3.5.2 After the tally

The auditor retrieves the election data $D$ and in particular the list $B$ of ballots and the result $r$. Then:

1. she checks consistency of $B$, that is, performs all the checks described at step 1 of section 3.5.1;
2. she checks that $B$ corresponds to the board monitored so far, thus performs all the checks described at step 2 of section 3.5.1;
3. she checks that the proofs of the result $r$ are valid w.r.t. $B$. She checks in particular the proofs of correct decryption and the proofs of correct shuffling (when shufflers have been used).

To ease verification of the trustees and the credential authorities, it is possible to display the hash of their public data (e.g. the public keys and the partial decryptions of the trustees, the hash of the list of the public credentials) in some human-readable form. In that case, the audit should also check that this human-readable data is consistent with the election data.

There is no tool support on the web interface for these checks, instead the command line tool verify can be used.

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 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 of size at least 14), encoded as a JSON string
- **I**: small integer, encoded as a JSON number
- **B**: boolean, encoded as a JSON boolean
- **N, Z, G**: big integer, written in base 10 and encoded as a JSON string

4.3 Common structures

\[
\begin{align*}
\text{proof} &= \{ \text{challenge} : \mathbb{Z}_q, \text{response} : \mathbb{Z}_q \} \\
\text{ciphertext} &= \{ \alpha : \mathbb{G}, \beta : \mathbb{G} \}
\end{align*}
\]

\footnote{Base58 characters are: 123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz}
4.4 Verification keys

\[
\begin{align*}
\text{public_key} &= G \\
\text{private_key} &= \mathbb{Z}_q \\
\text{trustee_public_key} &= \left\{ \begin{array}{l}
\text{pok} : \text{proof} \\
\text{public_key} : \text{public_key}
\end{array} \right\}
\end{align*}
\]

A private key is a number \( x \) modulo \( q \), chosen at random in the basic decryption mode, and computed after several interactions in the threshold mode. The corresponding \( \text{public_key} \) is \( X = g^x \). A \( \text{trustee_public_key} \) is a bundle of this public key with a \( \text{proof} \) of knowledge computed as follows:

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

where \( H_{\text{pok}} \) is computed as follows:

\[ H_{\text{pok}}(X, A) = \text{SHA256}(\text{pok} | X | A) \]

where \( \text{pok} \) 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 \( A = g^{\text{response}} / y^{\text{challenge}} \)
2. check that \( \text{challenge} = H_{\text{pok}}(\text{public_key}, A) \mod q \)

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 \( A \) as a proxy.

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

123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

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

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

\[
\text{signed_msg} = \left\{ \begin{array}{l}
\text{message} : \text{string} \\
\text{signature} : \text{proof}
\end{array} \right\}
\]

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

To verify a signature using a verification key \( \text{vk} \),
1. compute the commitment \( A = g^\text{response} \times \text{vk}^{\text{challenge}} \)
2. check that \( \text{challenge} = \text{SHA256}(\text{sigmsg} | M | A) \)

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

\[
\text{encrypted_msg} = \begin{cases} 
\text{alpha} & : \mathbb{G} \\
\text{beta} & : \mathbb{G} \\
\text{data} & : \text{string}
\end{cases}
\]

To compute the \text{encrypted_msg}:
1. pick random \( r, s \in \mathbb{Z}_q \)
2. compute \( \text{alpha} = g^r \)
3. compute \( \text{beta} = \text{ek}^r \times g^s \)
4. compute \( \text{data} \) as the hexadecimal encoding of the (symmetric) encryption of \( M \) using AES in CCM mode with \( \text{SHA256(key} | g^s) \) as the key and \( \text{SHA256(iv} | g^r) \) as the initialization vector (where numbers are written in base 10)

To decrypt an \text{encrypted_msg} using a decryption key \( \text{dk} \):
1. compute the symmetric key as \( \text{SHA256(key} \beta / (\text{alpha}^\text{dk})) \)
2. compute the initialization vector as \( \text{SHA256(iv} | \text{alpha}) \)
3. decrypt \( \text{data} \)

**4.5.2 Certificates**
A certificate is a \text{signed_msg} encapsulating a serialized \text{cert_keys} structure, itself filled with the public keys generated as described in section 4.5.1.

\[
\text{cert} = \text{signed_msg} \quad \text{cert_keys} = \begin{cases} 
\text{verfication} & : \mathbb{G} \\
\text{encryption} & : \mathbb{G}
\end{cases}
\]

The message is signed with the signing key associated to \text{verification}.

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

\[
\text{channel_msg} = \begin{cases} 
\text{recipient} & : \mathbb{G} \\
\text{message} & : \text{string}
\end{cases}
\]

When decoding such a message, \text{recipient} must be checked.
4.5.4 Polynomials

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

$$\text{polynomial} = \begin{cases} \text{polynomial} : & \text{string} \\ \text{secrets} : & \text{string}^* \\ \text{coefexps} : & \text{coefexps} \end{cases}$$

Suppose $T_i$ is the trustee who is computing. Therefore, $T_i$ knows the signing key $sk_i$ corresponding to $v_k$ and the decryption key $dk_i$ corresponding to $e_k$. $T_i$ first checks that keys indeed match. Then $T_i$ 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 $k = 0, \ldots, t$ and $s_{ij} = f_i(j) \mod q$ for $j = 1, \ldots, m$. $T_i$ then fills the polynomial structure as follows:

- the polynomial field is $\text{send}(sk_i, e_k, M)$ where $M$ is a serialized raw_polynomial structure

  $$\text{raw_polynomial} = \begin{cases} \text{polynomial} : & \mathbb{Z}_q^* \end{cases}$$

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

- the secrets field is $\text{send}(sk_i, e_k, M_1), \ldots, \text{send}(sk_i, e_k, M_m)$ where $M_{ij}$ is a serialized secret structure

  $$\text{secret} = \begin{cases} \text{secret} : & \mathbb{Z}_q \end{cases}$$

  filled with $s_{ij}$

- the coefexps field is a signed message containing a serialized raw_coefexps structure

  $$\text{coefexps} = \text{signed_msg} \quad \text{raw_coefexps} = \begin{cases} \text{coefexps} : & G^* \end{cases}$$

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

The public key of the election will be:

$$y = \prod_{z \in [1 \ldots m]} g^{f_z(0)} = \prod_{z \in [1 \ldots m]} A_{z0}$$

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.

$$\text{vinput} = \begin{cases} \text{polynomial} : & \text{string} \\ \text{secrets} : & \text{string}^* \\ \text{coefexps} : & \text{coefexps}^* \end{cases}$$

Suppose we are computing the vinput structure $v_{ij}$ for trustee $T_j$. We fill it as follows:

- the polynomial field is the same as the one of $P_j$

- the secret field is $\text{secret}(P_j)_j, \ldots, \text{secret}(P_m)_j$

- the coefexps field is $\text{coefexps}(P_j), \ldots, \text{coefexps}(P_m)$

Note that the coefexps field is the same for all trustees.

In step 5, $T_j$ checks consistency of $v_{ij}$ 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 $T_j$ receives $\Gamma$ and $v_i$, and produces a voutput $v_o_j$.

$$voutput = \{ \text{private_key : string} \} \begin{array}{l} \text{public_key : trustee_public_key} \end{array}$$

Trustee $T_j$ fills $v_o_j$ as follows:

- private_key is set to send$(s_{kj}, e_k, S_j)$, where $S_j$ is $T_j$’s (private) decryption key:

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

- public_key is set to a trustee_public_key structure built using $S_j$ as private key, which computes the corresponding public key and a proof of knowledge of $S_j$.

The administrator checks $v_o_j$ as follows:

- check that:

$$\text{public_key(public_key}(v_o_j)) = \prod_{i=1}^{m} \prod_{k=0}^{t} (A_{ik})^{s_k}$$

- check pok(public_key(v_o_j))

4.5.7 Threshold parameters

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

$$\text{threshold_parameters} = \begin{cases} \text{threshold} : I \\ \text{certs} : \text{cert}^* \\ \text{coexps} : \text{coexps}^* \\ \text{verification_keys} : \text{trustee_public_key}^* \end{cases}$$

The administrator fills it as follows:

- threshold is set to $t + 1$
- certs is set to $\Gamma = \gamma_1, \ldots, \gamma_m$
- coexps is set to the same value as the coexps field of vinputs
- verification_keys is set to public_key(v_o_1), ..., public_key(v_o_m)

4.6 Trustees

trustees = trustee_kind$^*$

trustee_kind = ["Single", trustee_public_key] | ["Pedersen", threshold_parameters]

A trustees structure is associated to each election. Such a structure is a list of either a single verification key as described in section 4.4 or threshold parameters as described in section 4.5.

Each item describes how a partial decryption is computed: either a specific (mandatory) verification key is used to compute a share, or a subset of a set of (optional) verification keys are used to compute a share.
The generality of this definition allows to mix mandatory and optional trustees during decryption. For example, in an election with 3 mandatory trustees, the trustees structure will look like:

\[
[["Single",...],["Single",...],["Single",...]]
\]

and in an election where only one trustee is mandatory, and a subset of another set of trustees (with a threshold) is needed to decrypt the result, will have a trustees structure that looks like:

\[
[["Single",...],["Pedersen",...]]
\]

4.7 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 \( s = \text{secret}(c) \) 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 \( \text{public}(c) = g^s \) is computed.

4.8 Questions

There are two types of questions: homomorphic ones and non-homomorphic ones. The difference is in the outcome of the election: with a homomorphic question, only the pointwise sum of all the answers (see 4.10) will be revealed at the end of the election whereas with a non-homomorphic question, each individual answer will be revealed.

4.8.1 Homomorphic questions

Homomorphic questions are represented directly (first alternative). They are the first type of question that was implemented in Belenos. They are suitable for many elections, like the ones where the voter is invited to select one choice among several (as in a referendum).

The blank field of \( \text{question}_h \) 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).
4.8.2 Non-homomorphic questions

Non-homomorphic questions are represented nested in a question_gen structure (second alternative), where the type property is set to NonHomomorphic, and the value property is set to a question_nh structure. They are needed when homomorphic questions are not suitable, for example when answers represent preferences or are too big.

4.9 Elections

\[
\text{election} = \begin{cases} 
\text{description} : \text{string} \\
\text{name} : \text{string} \\
\text{public_key} : \text{wrapped_pk} \\
\text{questions} : \text{question}^* \\
\text{uid} : \text{uuid} 
\end{cases}
\]

The election structure includes all public data related to an election and is sent to each voter. It uses the wrapped_pk defined below:

\[
\text{embedding} = \begin{cases} 
\text{padding} : \Pi \\
\text{bits_per_int} : \Pi 
\end{cases}
\]

\[
\text{group} = \begin{cases} 
\text{g} : \mathbb{G} \\
\text{p} : \mathbb{N} \\
\text{q} : \mathbb{N} \\
?\text{embedding} : \text{embedding} 
\end{cases}
\]

\[
\text{wrapped_pk} = \begin{cases} 
\text{group} : \text{group} \\
\text{y} : \mathbb{G} 
\end{cases}
\]

The election public key, which is denoted by \( y \) throughout this document, is computed during the setup phase, and bundled with the group parameters in a wrapped_pk structure. The embedding structure is required when the election includes a non-homomorphic question; its meaning will be explained in section 4.10.2.

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

- the election structure described above;
- the trustees structure described in section 4.6;
- the set \( L \) of public credentials.

4.10 Encrypted answers

\[
\text{answer}_h = \begin{cases} 
\text{choices} : \text{ciphertext}^* \\
\text{individual_proofs} : \text{iproof}^* \\
\text{overall_proof} : \text{iproof} \\
?\text{blank_proof} : \text{proof}^2 
\end{cases}
\]

\[
\text{answer}_n = \begin{cases} 
\text{choices} : \text{ciphertext} \\
\text{proof} : \text{proof} 
\end{cases}
\]

\[
\text{answer} = \text{answer}_h \mid \text{answer}_n
\]

The structure of an answer to a question depends on the type of the question. In all cases, a credential \( c \) is needed. Let \( s = \text{secret}(c) \), and \( S = g^s \) written in base 10.
4.10.1 Homomorphic answers

An answer to a homomorphic answer 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 additional 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. $\alpha = g^r$
3. $\beta = y^r g^m$

where $y$ is the election public key.

The individual proof that $m \in [0 \ldots 1]$ is computed by running `iprove(S,r,m,0,1)` (see section 4.11).

When a blank vote is not allowed, `overall_proof` proves that $M \in [\min \ldots \max]$ and is computed by running `iprove(S,R,M - \min,\min,\ldots,\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 additional proofs are as follows:

- `blank_proof` proves that $m_0 = 0 \lor m_\Sigma = 0$;
- `overall_proof` proves that $m_0 = 1 \lor m_\Sigma \in [\min \ldots \max]$.

They are computed as described in section 4.12.

4.10.2 Non-homomorphic answers

The plaintext answer to a non-homomorphic question is a vector $[v_1, \ldots, v_n]$ of small integers, one for each possible choice. When an election contains such a question, its `group` structure must include an embedding field, specifying how the vector of integers will be encoded into a single ciphertext:

- in the following, `bits_per_int` is denoted by $\kappa$ and `padding` by $p$;
- it is assumed that each $v_i$ is $\kappa$ bits (or less);
- $[v_1, \ldots, v_n]$ is encoded as:

$$
\xi = \text{group\_encode}_{\kappa,p}([v_1, \ldots, v_n]) = (((v_1 \times 2^\kappa + v_2) \times 2^\kappa + \cdots) \times 2^\kappa + v_n) \times 2^p + \varepsilon
$$

where $\varepsilon$ (of $p$ bits or less) is chosen so that $\xi \in G$;
- `choices` is set to an El Gamal encryption of $\xi$ as follows:

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

where $y$ is the election public key;
proof is computed as follows:
1. pick a random \( w \in \mathbb{Z}_q \)
2. compute \( A = g^w \)
3. challenge = \( H_{raweg}(S, y, \alpha, \beta, A) \)
4. response = \( w - r \times \text{challenge} \)

where \( H_{raweg} \) is computed as follows:
\[
H_{raweg}(S, y, \alpha, \beta, A) = \text{SHA256}(\text{raweg} | S | y, \alpha, \beta | A) \mod q
\]

where \( \text{raweg} \), 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.

The proof is verified as follows:
1. compute \( A = g^{\text{response}} \times \alpha^{\text{challenge}} \)
2. check that \( \text{challenge} = H_{raweg}(S, y, \alpha, \beta, A) \)

4.11 Proofs of interval membership

iproof = proof

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

1. for \(j \neq i\):
   (a) create \(\pi_j\) with a random \(\text{challenge}\) and a random \(\text{response}\)
   (b) compute
   \[
   A_j = g^{\text{response}} (\alpha^{\text{challenge}}) \quad \text{and} \quad B_j = (\beta / g^{M_j})^{\text{challenge}}
   \]
2. \(\pi_i\) 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) \(\text{challenge}(\pi_i) = H_{i\text{prove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) - \sum_{j \neq i} \text{challenge}(\pi_j) \mod q\)
   (d) \(\text{response}(\pi_i) = w + r \times \text{challenge}(\pi_i) \mod q\)

In the above, \(H_{i\text{prove}}\) is computed as follows:
\[
H_{i\text{prove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) = \text{SHA256}(\text{prove} | S | \alpha, \beta | A_0, B_0, \ldots, A_k, B_k) \mod q
\]
where \(\text{prove}\), 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 \(i\text{prove}(S, r, i, M_0, \ldots, M_k)\).

The proof is verified as follows:
1. for \(j \in [0 \ldots k]\), compute
   \[
   A_j = g^{\text{response}(\pi_j)} (\alpha^{\text{challenge}(\pi_j)}) \quad \text{and} \quad B_j = (\beta / g^{M_j})^{\text{challenge}(\pi_i)}
   \]
2. check that
   \[
   H_{i\text{prove}}(S, \alpha, \beta, A_0, B_0, \ldots, A_k, B_k) = \sum_{j \in [0 \ldots k]} \text{challenge}(\pi_j) \mod q
   \]
4.12 Proofs of possibly-blank votes

In this section, we suppose:

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

Note that \(\alpha_\Sigma, \beta_\Sigma\) and \(r_\Sigma\) can be easily computed from the encryptions of \(m_1, \ldots, m_n\) and their associated secrets.

Additionally, let \(P\) be the string \(g, y, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma\), where the commas are verbatim and the numbers are written in base 10. Let also \(M_1, \ldots, M_k\) be the sequence \(\min, \ldots, \max (k = \max - \min + 1)\).

4.12.1 Non-blank votes \((m_0 = 0)\)

**Computing blank_proof** In \(m_0 = 0 \lor m_\Sigma = 0\), the first case is true. The proof \(\text{blank\_proof}\) of the whole statement is the couple of proofs \((\pi_0, \pi_\Sigma)\) built as follows:

1. pick random \(\text{challenge}(\pi_\Sigma)\) and \(\text{response}(\pi_\Sigma)\) in \(\mathbb{Z}_q\)
2. compute \(A_\Sigma = g^{\text{response}(\pi_\Sigma) \times \alpha_\Sigma^{\text{challenge}(\pi_\Sigma)}}\) and \(B_\Sigma = g^{\text{response}(\pi_\Sigma) \times \beta_\Sigma^{\text{challenge}(\pi_\Sigma)}}\)
3. pick a random \(w\) in \(\mathbb{Z}_q\)
4. compute \(A_0 = g^w\) and \(B_0 = y^w\)
5. compute \(\text{challenge}(\pi_0) = \mathcal{H}_{\text{bproof0}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) - \text{challenge}(\pi_\Sigma) \mod q\)
6. compute \(\text{response}(\pi_0) = w - r_0 \times \text{challenge}(\pi_0) \mod q\)

In the above, \(\mathcal{H}_{\text{bproof0}}\) is computed as follows:

\[\mathcal{H}_{\text{bproof0}}(\ldots) = \text{SHA256}(\text{bproof0}|S|P|A_0, B_0, A_\Sigma, B_\Sigma) \mod q\]

where \(\text{bproof0}\), 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 \(m_0 = 1 \lor m_\Sigma \in [M_1, \ldots, M_k]\), 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 \(\text{challenge}(\pi_0)\) and \(\text{response}(\pi_0)\) in \(\mathbb{Z}_q\)
2. compute \(A_0 = g^{\text{response}(\pi_0) \times \alpha_0^{\text{challenge}(\pi_0)}}\) and \(B_0 = y^{\text{response}(\pi_0) \times (\beta_0/g)^{\text{challenge}(\pi_0)}}\)
3. for \(j > 0\) and \(j \neq i:\)
   (a) create \(\pi_j\) with a random \(\text{challenge}\) and a random \(\text{response}\) in \(\mathbb{Z}_q\)
   (b) compute \(A_j = g^{\text{response} \times \alpha_\Sigma^{\text{challenge}}}\) and \(B_j = g^{\text{response} \times (\beta_\Sigma/g^{M_j})^{\text{challenge}}}\)
4. pick a random \(w \in \mathbb{Z}_q\)
5. compute \(A_i = g^w\) and \(B_i = y^w\)
6. compute \(\text{challenge}(\pi_i) = \mathcal{H}_{\text{bproof}}(S, P, A_0, \ldots, A_k, B_k) - \sum_{j \neq i} \text{challenge}(\pi_j) \mod q\)
7. compute \( \text{response}(\pi_i) = w - r_\Sigma \times \text{challenge}(\pi_i) \mod q \)

In the above, \( H_{\text{bproof}_1} \) is computed as follows:

\[
H_{\text{bproof}_1}(\ldots) = \text{SHA256}(\text{bproof}_1|S|P|A_0, B_0, \ldots, A_k, B_k) \mod q
\]

where \( \text{bproof}_1 \), 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.

### 4.12.2 Blank votes \((m_0 = 1)\)

**Computing \text{blank\_proof}**  
In \( m_0 = 0 \lor m_\Sigma = 0 \), the second case is true. The proof \text{blank\_proof} of the whole statement is the couple of proofs \((\pi_0, \Sigma)\) built as in section 4.12.1 but exchanging subscripts 0 and \( \Sigma \) everywhere except in the call to \( H_{\text{bproof}_0} \).

**Computing \text{overall\_proof}**  
In \( m_0 = 1 \lor m_\Sigma \in [M_1 \ldots M_k] \), the first case is true. The proof of the whole statement is a \((k+1)\)-tuple \((\pi_0, \pi_1, \ldots, \pi_k)\) built as follows:

1. for \( j > 0 \):
   
   (a) create \( \pi_j \) with a random \text{challenge} and a random \text{response} in \( \mathbb{Z}_q \)
   
   (b) compute \( A_j = g^{\text{response}(\pi_j)} \times \alpha_\Sigma^{\text{challenge}(\pi_j)} \) and \( B_j = y^{\text{response}(\pi_j)} \times (\beta_\Sigma/g^{M_j})^{\text{challenge}(\pi_j)} \)

2. pick a random \( w \in \mathbb{Z}_q \)

3. compute \( A_0 = g^w \) and \( B_0 = y^w \)

4. compute

\[
\text{challenge}(\pi_0) = H_{\text{bproof}_1}(S, P, A_0, B_0, \ldots, A_k, B_k) - \sum_{j > 0} \text{challenge}(\pi_j) \mod q
\]

5. compute \( \text{response}(\pi_0) = w - r_0 \times \text{challenge}(\pi_0) \mod q \)

### 4.12.3 Verifying proofs

**Verifying \text{blank\_proof}**  
A proof of \( m_0 = 0 \lor m_\Sigma = 0 \) is a couple of proofs \((\pi_0, \Sigma)\) such that the following procedure passes:

1. compute \( A_0 = g^{\text{response}(\pi_0)} \times \alpha_0^{\text{challenge}(\pi_0)} \) and \( B_0 = y^{\text{response}(\pi_0)} \times \beta_0^{\text{challenge}(\pi_0)} \)

2. compute \( A_\Sigma = g^{\text{response}(\Sigma)} \times \alpha_\Sigma^{\text{challenge}(\Sigma)} \) and \( B_\Sigma = y^{\text{response}(\Sigma)} \times \beta_\Sigma^{\text{challenge}(\Sigma)} \)

3. check that

\[
H_{\text{bproof}_0}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) = \text{challenge}(\pi_0) + \text{challenge}(\Sigma) \mod q
\]

**Verifying \text{overall\_proof}**  
A proof of \( m_0 = 1 \lor m_\Sigma \in [M_1 \ldots M_k] \) is a \((k+1)\)-tuple \((\pi_0, \pi_1, \ldots, \pi_k)\) such that the following procedure passes:

1. compute \( A_0 = g^{\text{response}(\pi_0)} \times \alpha_0^{\text{challenge}(\pi_0)} \) and \( B_0 = y^{\text{response}(\pi_0)} \times (\beta_0/g)^{\text{challenge}(\pi_0)} \)

2. for \( j > 0 \), compute

\[
A_j = g^{\text{response}(\pi_j)} \times \alpha_j^{\text{challenge}(\pi_j)} \quad \text{and} \quad B_j = y^{\text{response}(\pi_j)} \times (\beta_j/g^{M_j})^{\text{challenge}(\pi_j)}
\]

3. check that

\[
H_{\text{bproof}_1}(S, P, A_0, B_0, \ldots, A_k, B_k) = \sum_{j=0}^{k} \text{challenge}(\pi_j) \mod q
\]
4.13 Signatures

\[
\text{signature} = \begin{cases} 
\text{public_key} & : \text{public_key} \\
\text{challenge} & : \mathbb{Z}_q \\
\text{response} & : \mathbb{Z}_q 
\end{cases}
\]

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. It is computed as follows:

1. compute \( s = \text{secret}(c) \)
2. pick a random \( w \in \mathbb{Z}_q \)
3. compute \( A = g^w \)
4. \( \text{public_key} = g^s \)
5. \( \text{challenge} = H_{\text{signature}}(\text{public_key}, A, \gamma_1, \ldots, \gamma_l) \mod q \)
6. \( \text{response} = w - s \times \text{challenge} \mod q \)

In the above, \( H_{\text{signature}} \) is computed as follows:

\[
H_{\text{signature}}(S, A, \gamma_1, \ldots, \gamma_l) = \text{SHA256}(\text{sig} | S | A | \text{alpha}(\gamma_1), \text{beta}(\gamma_1), \ldots, \text{alpha}(\gamma_l), \text{beta}(\gamma_l))
\]

where \( \text{sig} \), 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 \( A = g^{\text{response}} \times \text{public_key}^{\text{challenge}} \)
2. check that \( \text{challenge} = H_{\text{signature}}(\text{public_key}, A, \gamma_1, \ldots, \gamma_l) \mod q \)

4.14 Ballots

\[
\text{ballot} = \begin{cases} 
\text{answers} & : \text{answer} \\
\text{election_hash} & : \text{string} \\
\text{election_uid} & : \text{uuid} \\
\text{signature} & : \text{signature} 
\end{cases}
\]

The so-called hash (or fingerprint) of the election is computed with the function \( H_{\text{JSON}} \):

\[
H_{\text{JSON}}(J) = \text{BASE64}(\text{SHA256}(J))
\]

Where \( J \) is the serialization (done by the server) of the \text{election} structure.

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

4.15 Encrypted tally

\[
\text{ciphertexts}_h = \text{ciphertext}^* \quad \text{ciphertexts}_n = \text{ciphertext}^* \\
\text{encrypted_tally} = (\text{ciphertexts}_h | \text{ciphertexts}_n)^*
\]

A so-called \text{encrypted tally} is constructed out of the accepted ballots \( B_1, \ldots, B_n \). It is an array \([C_1, \ldots, C_m]\) where \( m \) is the number of questions. Each element \( C_i \) is itself an array of ciphertexts that is built differently depending on the type of the question:
• for homomorphic questions, each element of \( C_i \) (\texttt{ciphertexts}_h) is the pointwise product of the \( i \)-th ciphertext of all the ballots:

\[
C_{i,j} = \prod_k \text{choices(answers}(B_k)_j)
\]

where the product of two ciphertexts \((\alpha_1, \beta_1)\) and \((\alpha_2, \beta_2)\) is \((\alpha_1 \alpha_2, \beta_1 \beta_2)\);

• for non-homomorphic questions, \( C_i \) is directly made from the list of ciphertexts corresponding to the question:

\[
C_{i,k} = \text{choices(answers}(B_k)_i)
\]

In the end, in both cases, the encrypted tally is isomorphic to an array of arrays of ciphertexts:

\[
\text{encrypted_tally} \approx \text{ciphertext}^{**}
\]

### 4.16 Shuffles

If the election has non-homomorphic questions, let us say \( n \) out of \( m \) \((1 \leq n \leq m)\), non-homomorphic ciphertexts must be shuffled. They are first extracted from the encrypted tally \( a \):

\[
b = \text{nh_ciphertexts}(a) = [a_{i_1}, \ldots, a_{i_n}]
\]

where \( a \) is the \texttt{encrypted_tally} structure defined in 4.15. Conversely, once ciphertexts are shuffled as \( b' \) (see later), they must be merged into the encrypted tally as \( a' \) such that \( b' = \text{nh_ciphertexts}(a') \).

Shuffles are done in the same way as the CHVote system\(^2\). For each non-homomorphic question, its ciphertexts are re-encrypted and randomly permuted, and a zero-knowledge proof of the permutation is computed. All these shuffles are then assembled into a \texttt{shuffle} structure:

\[
\text{shuffle} = \{ \text{ciphertexts} : \text{ciphertext}^{**} \}
\]

which uses the following auxiliary types:

\[
\begin{align*}
\text{shuffle_commitment_rand} & = G \times G \times G \times (G \times G) \times G^* \\
\text{shuffle_response} & = \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^* \\
\text{shuffle_commitment_perm} & = G^* \\
\text{shuffle_chained_challenges} & = G^* \\
\text{shuffle_proof} & = \text{shuffle_commitment_rand} \\
& \quad \times \text{shuffle_response} \\
& \quad \times \text{shuffle_commitment_perm} \\
& \quad \times \text{shuffle_chained_challenges}
\end{align*}
\]

For each non-homomorphic question \( i \):

1. let \( e = b_i = [e_1, \ldots, e_N] \) be the array of ciphertexts corresponding to question \( i \) \((N\) being the number of ballots);
2. let \((e', r', \psi) = \text{GenShuffle}(e, y)\) \((y\) being the public key of the election);
3. let \( \pi = \text{GenShuffleProof}(e, e', r', \psi, y) \);
4. set \text{ciphertexts} to \( e' \) and \text{proofs} to \( \pi \).

The functions \text{GenShuffle} and \text{GenShuffleProof} are the same as in CHVote and are given in section\(^6\).

Typically, several shuffles will be computed sequentially by different persons.

\(^2\)See version 1.3.2 of the CHVote System Specification at \[8\]
4.17 Partial decryptions

\[
\text{partial\_decryption} = \{ \text{decryption\_factors} : G^{**}, \text{decryption\_proofs} : \text{proof}^{**} \}
\]

From the encrypted tally \(a'\) (where answers to non-homomorphic questions have been shuffled), 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:

\[
\text{decryption\_factors}_{i,j} = \alpha(a'_{i,j})^x
\]

and proofs that they were correctly computed. Each decryption\_proofs\_[i,j] is computed as follows:

1. pick a random \(w \in \mathbb{Z}_q\)
2. compute \(A = g^w\) and \(B = \alpha(a'_{i,j})^w\)
3. challenge = \(H_{\text{decrypt}}(X, A, B)\)
4. response = \(w + x \times \text{challenge} \mod q\)

In the above, \(H_{\text{decrypt}}\) is computed as follows:

\[
H_{\text{decrypt}}(X, A, B) = \text{SHA256}(\text{decrypt}\_|X\_A\_B) \mod q
\]

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 \(A = \frac{g^{\text{response}}}{\text{public\_key}(k)\text{challenge}}\) and \(B = \frac{\alpha(a'_{i,j})^{\text{response}}}{\text{decryption\_factors}_{i,j}^{\text{challenge}}}\)
2. check that \(H_{\text{decrypt}}(\text{public\_key}(k), A, B) = \text{challenge}\)

4.18 Election result

\[
\text{result} = \{ \text{num\_tallied} : II, \text{encrypted\_tally} : \text{encrypted\_tally}, \text{?shuffles} : \text{shuffle}^*, \text{partial\_decryptions} : \text{partial\_decryption}^*, \text{result} : (\mathbb{I}^* | \mathbb{I}^{**})^* \}
\]

The encrypted\_tally field is set to the encrypted tally \(a'\).

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

\[
F_{i,j} = \prod_{z \in [1..m]} \text{partial\_decryptions}_{z,i,j}
\]

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

\[
F_{i,j} = \prod_{z \in I} (\text{partial\_decryptions}_{z,i,j})^{\lambda_z^T}
\]

where \(I = \{z_1, \ldots, z_{t+1}\}\) is the set of indexes of supplied partial decryptions, and \(\lambda_z^T\) are the Lagrange coefficients:

\[
\lambda_z^T = \prod_{k \in I \setminus \{z\}} \frac{k}{k - z} \mod q
\]

The result field of the result structure is then computed as follows:
• if question $i$ is homomorphic,

$$\text{result}_{i,j} = \log_g \left( \frac{\beta(a'_{i,j})}{F_{i,j}} \right)$$

where $j$ represents an answer. The discrete logarithm can be easily computed because it is bounded by $\text{num\_tallied}$;

• if question $i$ is non-homomorphic,

$$\text{result}_{i,j} = \text{group\_decode}_{\kappa,p} \left( \frac{\beta(a'_{i,j})}{F_{i,j}} \right)$$

where $j$ represents a ballot, and $\text{group\_decode}$ is the inverse of $\text{group\_encode}$ from section 4.10.2.

If the election has non-homomorphic questions, the $\text{shuffles}$ field is set to the computed $\text{shuffle}$ structures; otherwise, it is absent.

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

• the $\text{election}$ structure;

• all the $\text{trustee\_public\_keys}$, or the $\text{threshold\_parameters}$, that were generated during the $\text{setup\_phase}$

• the set of public credentials;

• the set of ballots;

• the $\text{result}$ structure described above.

5 Default group parameters

5.1 Homomorphic group

This group is optimized for elections that have only homomorphic questions and is used in this case. Its parameters have been generated by the $\text{fips\_sage}$ script (available in Belenios sources), which is itself based on FIPS 186-4.
\[ p = 20694785691422546 \]
\[ 40101364365750080864922989925711040997100884787057374219242 \]
\[ 71740192223725494768433812906663380789584094960054389638289 \]
\[ 79639308377390572980360597387429678137677751899858872735865 \]
\[ 04998116099310535887780980030790491654067717614198678527 \]
\[ 27347447634185360003569830519314428456170191100078673730733 \]
\[ 56412397132897913240474578834468260652327974647951137672658 \]
\[ 6935821800463179220736688600526271863633868088796882120769432 \]
\[ 36614949100292344434637322145884100586421050242120365433561 \]
\[ 2013204811188524087310770141516662001623131716937189248078 \]
\[ 507711827842317498073276598828825169183103125680162072880719 \]
\[ g = 2402352677501852 \]
\[ 2092276877035323999327122876573878364916510075318787663274146 \]
\[ 353219320285676155269678799694668298743988095083896573425601 \]
\[ 90060106847716491735474137283104610458681341451781646755400 \]
\[ 527402889846139864532661215055797097162016168270312886432456 \]
\[ 6638348636357821061549184199825343415189740658186868661151358 \]
\[ 57641013888221539601604322884360393098933366277284806593138 \]
\[ 406010231675057637779826651036068224066350766797746025346253 \]
\[ 773085133173495194248967754052573659049492477631475911575198 \]
\[ 77547712114312904927486000205478127054728284104972518639858334 \]
\[ 1157005683536955534237814755824918960509296880037745308460627 \]
\[ q = 78571373251071885 \]
\[ 0799276598126714501218212412258408794611510081919805623223441 \]

The additional output of the generation algorithm is:

\[
\begin{align*}
\text{domain\_parameter\_seed} & = 478953892617249466 \\
& = 166106476098847629563138168027 \\
& = 716882488732447198349000396592 \\
& = 020632875172724552145560167746
\end{align*}
\]

\[
\begin{align*}
\text{counter} & = 109
\end{align*}
\]

### 5.2 Non-homomorphic group

The group described in the previous section is not suitable for encoding non-homomorphic answers (the `group_encode` function of section 4.10.2). Therefore, we use a different group if the election
has non-homomorphic questions. This group is the 2048-bit one defined in RFC 3526:

\[
p = 3231700607131100760338913926423828248817941241140239112842009751400741706634354226196894173635693471790173790970419175460587320919502885375896185622153212175412514901774752072035796078236248884346189477586410592864009411723245426622522193230540919037680524235519125679715870117001058055877651038861847280257976054903569732561526167081339361799541336476559160368317896729073178384589680639671900977202194168647225871031411336429319536193471636533209717077448227988588565369208645296636077250268955505928362751121174096972998068410554359584866583291642136218231078990099448652468262416972035911852507045361090559
\]

Additionally, its embedding field is set to:

\[
\{ \text{padding} = 8 \}
\]

6 Shuffle algorithms

The algorithms GenShuffle and GenShuffleProof are referred to in section 4.16. They were taken from version 1.3.2 of the CHVote System Specification [8], and are given here for self-completeness. We also give the CheckShuffleProof algorithm, used to check a proof produced by GenShuffleProof. For more explanations on these algorithms, please refer to the CHVote System Specification.
Input
- \( e = [e_1, \ldots, e_N] \in \text{ciphertext}^N \): encrypted answers to one non-homomorphic question
- \( y \in \mathbb{G} \): public key of the election

Algorithm
1. \( \psi \leftarrow \text{GenPermutation}(N) \) \hspace{1cm} // \( \psi = [j_1, \ldots, j_N] \), see table 2
2. For \( i = 1, \ldots, N \):
   - \( (e'_i, r'_i) \leftarrow \text{GenReEncryption}(e_i, y) \) \hspace{1cm} // see table 3
3. \( e' \leftarrow [e'_{j_1}, \ldots, e'_{j_N}] \)
4. \( r' \leftarrow [r'_1, \ldots, r'_N] \)
5. Return \( (e', r', \psi) \) \hspace{1cm} // \( e' \in \text{ciphertext}^N, r' \in \mathbb{Z}_q^N, \psi \in \Psi_N \)

Table 1: Function \text{GenShuffle}(e, y)

Input
- \( N \in \mathbb{N} \): permutation size

Algorithm
1. \( I \leftarrow [1, \ldots, N] \)
2. For \( i = 0, \ldots, N - 1 \):
   - \( \text{(a)} \) Pick \( k \) uniformly at random in \( \{i, \ldots, N - 1\} \)
   - \( \text{(b)} \) \( j_{i+1} \leftarrow I[k] \)
   - \( \text{(c)} \) \( I[k] \leftarrow I[i] \)
3. \( \psi \leftarrow [j_1, \ldots, j_N] \)
4. Return \( \psi \) \hspace{1cm} // \( \psi \in \Psi_N \)

Table 2: Function \text{GenPermutation}(N)
Input

- \( e \in \text{ciphertext} \): one encrypted answer to one non-homomorphic question
- \( y \in \mathbb{G} \): public key of the election

Algorithm

1. Pick \( r' \) uniformly at random in \( \mathbb{Z}_q \)
2. \( \alpha' \leftarrow \text{alpha}(e) \times g^{r'} \)
3. \( \beta' \leftarrow \text{beta}(e) \times y^{r'} \)
4. Let \( e' \) be a new ciphertext with \( \alpha = \alpha' \) and \( \beta = \beta' \)
5. Return \( (e', r') \)

Table 3: Function \( \text{GenReEncryption}(e, y) \)
Input

- $e = [e_1, \ldots, e_N] \in \text{ciphertext}^N$: encrypted answers to one question; we will denote by $\alpha_i$ and $\beta_i$ the contents of $e_i$
- $e' = [e'_1, \ldots, e'_N] \in \text{ciphertext}^N$: shuffled encrypted answers; we will denote by $\alpha'_i$ and $\beta'_i$ the contents of $e'_i$
- $r' = [r'_1, \ldots, r'_N] \in \mathbb{Z}_2^N$: re-encryption randomizations
- $\psi = [j_1, \ldots, j_N] \in \Psi_N$: permutation
- $pk \in G$: the public key of the election

Algorithm

1. $h \leftarrow \text{GetSecondaryGenerator}(), h \leftarrow \text{GetGenerators}(N)$ \hfill // see tables 6 and 7
2. $(c, r) \leftarrow \text{GenPermutationCommitment}(\psi, h)$ \hfill // see table 9
3. $\text{str} \leftarrow [c][c'][c]$ \hfill // see table 10
4. $u \leftarrow \text{NIKZPChallenges}(N, \text{shuffle-challenges} | \text{str}_e)$ \hfill // see table 11
5. For $i = 1, \ldots, N$: $u'_i \leftarrow u_i$
6. $u' \leftarrow [u'_1, \ldots, u'_N]$
7. $(c, r) \leftarrow \text{GenCommitmentChain}(h, u')$ \hfill // see table 12
8. For $i = 1, \ldots, 4$: pick $\omega_i$ at random in $\mathbb{Z}_q$
9. For $i = 1, \ldots, N$: pick $\omega_i'$ at random in $\mathbb{Z}_q$
10. $t_1 \leftarrow g^{\omega_1}, t_2 \leftarrow g^{\omega_2}, t_3 \leftarrow g^{\omega_3} \prod_{i=1}^N h_i^{\omega_i}$
11. $(t_{4,1}, t_{4,2}) \leftarrow (pk^{-\omega_4} \prod_{i=1}^N (\beta'_i)^{\omega_i'}, g^{-\omega_4} \prod_{i=1}^N (\alpha'_i)^{\omega_i'})$
12. $\check{c_0} \leftarrow h$
13. For $i = 1, \ldots, N$: $\check{c}_i \leftarrow g^{\omega_i} c_{i-1}\check{c}_{i-1}$
14. $t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\check{c}_1, \ldots, \check{c}_N]), \text{str}_t \leftarrow [[t_1, t_2, t_3, t_{4,1}, t_{4,2}][[\check{c}_1, \ldots, \check{c}_N]]$
15. $y \leftarrow (e, e', c, \check{c}, \text{str}_t), \text{str}_y \leftarrow \text{str}_t, \text{pk}$ \hfill // $pk$ taken as a number in base 10
16. $c \leftarrow \text{NIKZPChallenge}(\text{shuffle-challenge}_t, \text{str}_t, \text{str}_y)$ \hfill // see table 13
17. $r \leftarrow \sum_{i=1}^N r_i \mod q, s_1 \leftarrow \omega_1 + c \times r \mod q$
18. $v_N \leftarrow 1$
19. For $i = N - 1, \ldots, 1$: $v_i \leftarrow u_{i+1} v_{i+1} \mod q$
20. $\check{r} \leftarrow \sum_{i=1}^N \check{r}_i u_i \mod q, s_2 \leftarrow \omega_2 + c \times \check{r} \mod q$
21. $r \leftarrow \sum_{i=1}^N r_i u_i \mod q, s_3 \leftarrow \omega_3 + c \times r \mod q$
22. $r' \leftarrow \sum_{i=1}^N r'_i u_i \mod q, s_4 \leftarrow \omega_4 + c \times r' \mod q$
23. For $i = 1, \ldots, N$: $s_i \leftarrow \omega_i + c \times r_i \mod q, s'_i \leftarrow \omega'_i + c \times u'_i \mod q$
24. $s \leftarrow (s_1, s_2, s_3, s_4, [s'_1, \ldots, s'_N], [s'_1, \ldots, s'_N])$
25. $\pi \leftarrow (t, s, c, \check{c})$ \hfill // $\pi \in \text{shuffle_proof}$
26. Return $\pi$

Table 4: Function $\text{GenShuffleProof}(e, e', r', \psi, pk)$
Input

- \( \pi \in \text{shuffle} \_\text{proof} \): shuffle proof
- \( e = [e_1, \ldots, e_N] \in \text{ciphertext}^N \): encrypted answers to one question; we will denote by \( \alpha_i \) and \( \beta_i \) the contents of \( e_i \)
- \( e' = [e'_1, \ldots, e'_N] \in \text{ciphertext}^N \): shuffled encrypted answers; we will denote by \( \alpha'_i \) and \( \beta'_i \) the contents of \( e'_i \)
- \( pk \in G \): the public key of the election

Algorithm

1. \((t, s, c, \hat{c}) \leftarrow \pi \)
2. \((t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\hat{t}_1, \ldots, \hat{t}_N]) \leftarrow t \)
3. \((s_1, s_2, s_3, [s_1, \ldots, s_N], [s'_1, \ldots, s'_N]) \leftarrow s \)
4. \([c_1, \ldots, c_N] \leftarrow c, [\hat{c}_1, \ldots, \hat{c}_N] \leftarrow \hat{c} \)
5. \(h \leftarrow \text{GetSecondaryGenerator}(), h \leftarrow \text{Generators}(N)\) \hspace{1cm} // see tables 6 and 7
6. \(\text{str}_c \leftarrow [e][e'][c]\) \hspace{1cm} // see table 10
7. \(\text{str}_\alpha \leftarrow [[t_1, t_2, t_3, (t_{4,1}, t_{4,2})][[\hat{t}_1, \ldots, \hat{t}_N]]\)
8. \(\text{str}_s \leftarrow [\text{str}_c \text{shuffle-challenge}][\text{str}_r, \text{str}_s] \) \hspace{1cm} // see table 11
9. \(\text{str}_r \leftarrow [\text{str}_c][pk] \) \hspace{1cm} // \( pk \) taken as a number in base 10
10. \(\text{e} \leftarrow \text{GetNIZKPChallenge}(\text{shuffle} \_\text{challenge} \text{str}_r \text{str}_s) \) \hspace{1cm} // see table 13
11. \(\hat{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i \)
12. \(u \leftarrow \prod_{i=1}^N u_i \mod q \)
13. \(\hat{c} \leftarrow h^u \)
14. \(\hat{c} \leftarrow \hat{c}_N / h^u \)
15. \(\hat{c} \leftarrow \prod_{i=1}^N c_i^{\alpha_i} \)
16. \((\alpha', \beta') \leftarrow (\prod_{i=1}^N \alpha_i^{u_i}, \prod_{i=1}^N \beta_i^{u_i}) \)
17. \(t'_1 \leftarrow \hat{c}^{-c} \times g^{t_1} \)
18. \(t'_2 \leftarrow \hat{c}^{-c} \times g^{t_2} \)
19. \(t'_3 \leftarrow \hat{c}^{-c} \times g^{t_3} \prod_{i=1}^N h_i^{x_i} \)
20. \((t'_{4,1}, t'_{4,2}) \leftarrow ((\beta')^{-c} \times pk^{-s_4} \prod_{i=1}^N (\beta'_i)^{t'_i}, (\alpha')^{-c} \times g^{-s_4} \prod_{i=1}^N (\alpha'_i)^{t'_i}) \)
21. For \( i = 1, \ldots, N \): \( \hat{t}_i \leftarrow \hat{c}_i^{c} \times g^{t_i} \times \hat{c}_i^{x_i} \)
22. Return \( (t_1 = t'_1) \land (t_2 = t'_2) \land (t_3 = t'_3) \land (t_{4,1} = t'_{4,1}) \land (t_{4,2} = t'_{4,2}) \land \prod_{i=1}^N (\hat{t}_i = \hat{t}_i) \)

Table 5: Function \( \text{CheckShuffleProof}(\pi, e, e', pk) \)
Algorithm

1. \( h \leftarrow \text{GetGenerator}(-1) \)  
   \[ \text{// see table 8} \]
2. Return \( h \)  
   \[ \text{// } h \in \mathbb{G}^N \]

Table 6: Function GetSecondaryGenerator()

---

Input

- \( N \in \mathbb{N} \): number of independent generators to get

Algorithm

1. For \( i = 0, \ldots, N - 1 \): \( h_i \leftarrow \text{GetGenerator}(i) \)  
   \[ \text{// see table 8} \]
2. \( h \leftarrow [h_0, \ldots, h_{N-1}] \)
3. Return \( h \)  
   \[ \text{// } h \in \mathbb{G}^N \]

Table 7: Function GetGenerators(\( N \))
Input
• \( i \in \mathbb{Z} \): number of the independent generator to get

State (shared between all runs)
• \( \mathcal{X} \in \mathcal{P}(\mathbb{N} \times \mathbb{G}) \) (initialized to \( \emptyset \)): generators to avoid

Algorithm
1. \( c \leftarrow (p - 1)/q \) \hspace{1cm} // typically, \( c = 2 \)
2. \( x \leftarrow \text{SHA256}(\text{ggen}\mid i) \) \hspace{1cm} // \( i \) in base 10, output as a big-endian number
3. \( h \leftarrow x^c \)
4. If \( h \in \{0, 1, g\} \), abort
5. If \( \exists j \neq i, (j, h) \in \mathcal{X} \), abort
6. \( \mathcal{X} \leftarrow \mathcal{X} \cup \{(i, h)\} \)
7. Return \( h \) \hspace{1cm} // \( h \in \mathbb{G} \)

Table 8: Function GetGenerator\((i)\) (for a multiplicative subgroup of a finite field)

Input
• \( \psi = \left[ j_1, \ldots, j_N \right] \in \Psi_N \): permutation
• \( h = \left[ h_1, \ldots, h_N \right] \in \mathbb{G}^N \): independent generators

Algorithm
1. For \( i = 1, \ldots, N \):
   • Pick \( r_{j_i} \) at random in \( \mathbb{Z}_q \)
   • \( c_{j_i} \leftarrow g^{r_{j_i}} \times h_i \)
2. \( c \leftarrow [c_1, \ldots, c_N] \)
3. \( r \leftarrow [r_1, \ldots, r_N] \)
4. Return \((c, r)\) \hspace{1cm} // \( c \in \mathbb{G}^N, r \in \mathbb{Z}_q^N \)

Table 9: Function GenPermutationCommitment\((\psi, h)\)
Input

- \( e = [e_1, \ldots, e_N] \in \text{ciphertext}^N \): array of ciphertexts, or
- \( c = [c_1, \ldots, c_N] \in \mathbb{G}^N \): array of group elements

Algorithm
1. set \( S \) to the empty string
2. For \( i = 1, \ldots, N \):
   - append \( \alpha(e_i) \), a comma, \( \beta(e_i) \) and a comma to \( S \), or \( // \) in base 10
   - append \( c_i \) and a comma to \( S \) \( // \) in base 10
3. Return \( S \) \( // S \in \text{string} \)

Table 10: Functions \([e]\) and \([c]\)

Input

- \( N \in \mathbb{N} \): number of ciphertexts
- \( S \in \text{string} \): challenge string

Algorithm
1. \( H \leftarrow \text{SHA256}(S) \) \( // \) output interpreted as an hexadecimal string
2. For \( i = 0, \ldots, N - 1 \):
   (a) \( T \leftarrow \text{SHA256}(i) \) \( // \) input taken as decimal, output interpreted as hexadecimal
   (b) \( u_i \leftarrow \text{SHA256}(HT) \mod q \) \( // \) output interpreted as big-endian
3. \( \mathbf{u} \leftarrow [u_0, \ldots, u_{N-1}] \)
4. Return \( \mathbf{u} \) \( // \mathbf{u} \in \mathbb{Z}_q^N \)

Table 11: Function GetNiZPKChallenges\((N, S)\)
Input

- $c_0 \in \mathbb{G}$: initial commitment
- $u = [u_1, \ldots, u_N] \in \mathbb{Z}_q^N$: public challenges

Algorithm

1. For $i = 1, \ldots, N$:
   (a) Pick $r_i$ at random in $\mathbb{Z}_q$
   (b) $c_i \leftarrow g^{r_i} \times c_0^{u_i-1}$
2. $\mathbf{c} \leftarrow [c_1, \ldots, c_N]$
3. $\mathbf{r} \leftarrow [r_1, \ldots, r_N]$
4. Return $(\mathbf{c}, \mathbf{r})$  

// $\mathbf{c} \in \mathbb{G}^N$, $\mathbf{r} \in \mathbb{Z}_q^N$

Table 12: Function GenCommitmentChain($c_0, u$)

Input

- $S \in \text{string}$: challenge string

Algorithm

1. $c \leftarrow \text{SHA256}(S) \mod q$  
   // output interpreted as a big-endian number
2. Return $c$

// $c \in \mathbb{Z}_q$

Table 13: Function GetNIZPKChallenge($S$)

30
References


