Belenios specification

Stéphane Glondu

Version 1.17

Contents

1 Introduction **2**

2 Parties **3**

3 Processes **3**
3.1 Election setup . . . 3
3.1.1 Filling in the `trustees` structure . . . 4
3.2 Vote . . . 5
3.3 Credential recovery . . . 5
3.4 Tally . . . 5
3.5 Audit . . . 6
3.5.1 During the voting phase . . . 6
3.5.2 After the tally . . . 6

4 Messages **6**
4.1 Conventions . . . 6
4.2 Basic types . . . 7
4.3 Common structures . . . 7
4.4 Verification keys . . . 7
4.5 Messages specific to threshold decryption support . . . 7
4.5.1 Public key infrastructure . . . 7
4.5.2 Certificates . . . 9
4.5.3 Channels . . . 9
4.5.4 Polynomials . . . 9
4.5.5 Vinputs . . . 10
4.5.6 Voutputs . . . 10
4.5.7 Threshold parameters . . . 10
4.6 Trustees . . . 11
4.7 Credentials . . . 11
4.8 Questions . . . 12
4.8.1 Homomorphic questions . . . 12
4.8.2 Non-homomorphic questions . . . 12
4.9 Elections . . . 12
4.10 Encrypted answers . . . 13
4.10.1 Homomorphic answers . . . 13
4.10.2 Non-homomorphic answers . . . 14
4.11 Proofs of interval membership . . . 15
4.12 Proofs of possibly-blank votes . . . 15
4.12.1 Non-blank votes ($m_0 = 0$) . . . 16
4.12.2 Blank votes ($m_0 = 1$) . . . 17
4.12.3 Verifying proofs . . . 17

4.13 Signatures 18
4.14 Ballots 18
4.15 Encrypted tally 19
4.16 Shuffles 19
4.17 Partial decryptions 20
4.18 Election result 20

5 Group parameters **22**
5.1 BELENIOS-2048 22
5.2 RFC-3526-2048 22

6 Shuffle algorithms **23**

1 Introduction

References. This document is a specification of the voting protocol implemented in Belenios 1.17 . 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 is spread across several references.

- The threshold decryption scheme is based on a "folklore" scheme: Pedersen's [10] 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 [8]. Additionally, ballots are signed to avoid ballot stuffing, as introduced in [5] and also described in [6]. Zero-knowledge proofs include the complete description of the group to avoid attacks described in [7].
- 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 [9].

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 $\mathbb{G}$ 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.

Weights. In the homomorphic case (and only in the homomorphic case), each voter has a weight: a ballot is counted as many times as the weight of its owner. Usually, the weight of all voters is 1 but sometimes, it may be useful to assign different weights. We assume the sum of all weights is not too big, so that it can be computed as the discrete logarithm of some group element.

2 Parties

- $\mathcal{A}$: server administrator
- $\mathcal{C}$: credential authority
- $\mathcal{T}_1, \dots, \mathcal{T}_m$: trustees
- $\mathcal{V}_1, \dots, \mathcal{V}_n$: voters; each voter has a weight w_i equal to 1 by default
- $\mathcal{S}$: voting server
  The voting server maintains the public data D that consists of:
  - the election data E
  - the structure PK that contains the verification keys of the trustees and other verification material
  - the list L of public credentials
  - the list B of accepted ballots
  - the result of the election `result` (once the election is tallied)

3 Processes

3.1 Election setup

1. $\mathcal{A}$ starts the preparation of an election, providing in particular the questions and a list of voter
2. $\mathcal{S}$ generates a fresh `uuid` u and sends it to $\mathcal{C}$
3. $\mathcal{C}$ generates credentials $c_1, \dots, c_n$ and computes
   $$L = \mathsf{shuffle}((\mathsf{public}(c_1), w_1), \dots, (\mathsf{public}(c_n), w_n))$$
4. for $j \in [1 \dots n]$, $\mathcal{C}$ sends c_j to $\mathcal{V}_j$
5. (optional) $\mathcal{C}$ forgets $c_1, \dots, c_n$
6. $\mathcal{C}$ sends L to $\mathcal{S}$
7. $\mathcal{S}$ checks that the multiset of weights in L is the same as $\{w_1, \dots, w_n\}$
8. $\mathcal{S}$ defines the shape of the `trustees` structure that will be used in the election depending on $\mathcal{A}$'s instructions;
9. $\mathcal{S}$ and $\mathcal{T}_1, \dots, \mathcal{T}_m$ run key establishment protocols (see 3.1.1) as needed to fill in the `trustees` structure;
10. $\mathcal{S}$ creates the `election` E
11. $\mathcal{S}$ loads E and L
12. $\mathcal{C}$ checks that the list of public credentials L is exactly the one that appears on the election data of the election of `uuid` u.

Step 5 is optional. It offers a better protection against ballot stuffing in case $\mathcal{C}$ unintentionally leaks private credentials.

3.1.1 Filling in the `trustees` structure

The `trustees` structure consists of `"Single"` or `"Pedersen"` items. For each of these items, one or several trustees run the corresponding protocol below to produce a sub-key $y_\mathcal{T}$. Once all protocols have been run, $\mathcal{S}$ synthesizes the global election public key y from the sub-keys computed in each protocol by multiplying them:

$$y = \prod_{\mathcal{T}} y_{\mathcal{T}}$$

`"Single"` protocol This protocol involves a single trustee $\mathcal{T}$, whose presence will be required to compute the tally.

1. $\mathcal{T}$ generates a `trustee_public_key` γ and sends it to $\mathcal{S}$
2. $\mathcal{S}$ checks γ

Later, when the election is open:

1. $\mathcal{T}$ checks that γ appears in the set of verification keys PK of the election of `uuid` u (the id of the election should be publicly known)

The sub-key for this protocol is the public_key field of γ.

`"Pedersen"` protocol This protocol involves μ trustees $\mathcal{T}_1, \ldots, \mathcal{T}_\mu$ such that only a subset of $t+1$ of them will be needed to compute the tally.

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

Later, when the election is open:

1. for $z \in [1 \ldots \mu]$, $\mathcal{T}_z$ checks that γ_z appears in the set of verification keys PK of the election of `uuid` u (the id of the election should be publicly known).

The sub-key for this protocol is computed from the polynomials of each trustee as specified in section 4.5.4.

3.2 Vote

1. $\mathcal{V}$ gets E
2. $\mathcal{V}$ creates a `ballot` b and submits it to $\mathcal{S}$
3. $\mathcal{S}$ processes b:
   (a) let C be the public credential used in b (its $\mathsf{credential}$ field)
   (b) $\mathcal{S}$ checks that the weight of C and the weight of $\mathcal{V}$ agree
   (c) $\mathcal{S}$ checks that $C \in L$ and C has not been used in a ballot cast by another voter
   (d) (revote case) if $\mathcal{V}$ has already voted, $\mathcal{S}$ checks that it was with C
   (e) $\mathcal{S}$ checks all zero-knowledge proofs of b
   (f) $\mathcal{S}$ adds b to B (or replaces the old ballot in case of revote)
4. at any time (even after tally), $\mathcal{V}$ may check that b appears in the list of accepted ballots B and the weight of her ballot as it appears in B is equal to her weight

3.3 Credential recovery

If $\mathcal{C}$ has forgotten the private credentials of the voter (optional step 5 of the setup) then credentials cannot be recovered.

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

1. $\mathcal{V}_i$ contacts $\mathcal{C}$
2. $\mathcal{C}$ looks up $\mathcal{V}_i$'s private credential c_i
3. $\mathcal{C}$ sends c_i

3.4 Tally

1. $\mathcal{A}$ stops $\mathcal{S}$ and $\mathcal{S}$ computes the initial `encrypted_tally` Π_0
2. $\mathcal{S}$ extracts the non-homomorphic ciphertexts from the encrypted tally (see section 4.16):
$$\tilde{\Pi}_0 = \mathsf{nh\_ciphertexts}(\Pi_0)$$
3. if the election contains a non-homomorphic part, that is, if $\tilde{\Pi}_0 \neq []$, then for $z \in [1 \dots m]$:
   (a) $\mathcal{S}$ sends $\tilde{\Pi}_{z-1}$ to $\mathcal{T}_z$
   (b) $\mathcal{T}_z$ runs the shuffle algorithm, producing a `shuffle` σ_z and sends it to $\mathcal{S}$
   (c) $\mathcal{S}$ verifies σ_z and extracts $\tilde{\Pi}_z = \mathsf{ciphertexts}(\sigma_z)$
4. $\mathcal{S}$ merges shuffled non-homomorphic ciphertexts with homomorphic ciphertexts, i.e. builds Π such that:
$$\tilde{\Pi}_m = \mathsf{nh\_ciphertexts}(\Pi)$$
5. for $z \in [1 \dots m]$ (or, if in threshold mode, a subset of it of size at least $t+1$),
   (a) $\mathcal{S}$ 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 $\mathcal{S}$
   (c) $\mathcal{S}$ verifies δ_z
6. $\mathcal{S}$ combines all the partial decryptions, computes and publishes the election `result`
7. $\mathcal{T}_z$ checks that δ_z and σ_z appears in `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 also checks that the weights correspond;
   - 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 `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 $\mathsf{field}(o)$ to access the field field of o.

4.2 Basic types

- `string`: JSON string
- `uuid`: election identifier (a string of Base58 characters[1] 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

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

4.4 Verification keys

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

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

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 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. $\textsf{challenge} = \mathcal{H}_{\textsf{pok}}(X, A) \mod q$
4. $\textsf{response} = w - x \times \textsf{challenge} \mod q$

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

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

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

1. compute $A = g^{\textsf{response}} \times X^{\textsf{challenge}}$
2. check that $\textsf{challenge} = \mathcal{H}_{\textsf{pok}}(X, A) \mod q$

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 $\mathcal{S}$ as a proxy.

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

`123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

[1] Base58 characters are: `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.

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

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).

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

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.

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

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

  $$\texttt{raw\_polynomial} = \left\{ \textsf{polynomial} \quad : \quad \mathbb{Z}_q^* \right\}$$

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

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

  $$\texttt{secret} = \left\{ \textsf{secret} \quad : \quad \mathbb{Z}_q \right\}$$

  filled with s_{ij}

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

  $$\texttt{coefexps} = \texttt{signed\_msg} \qquad \texttt{raw\_coefexps} = \left\{ \textsf{coefexps} \quad : \quad \mathbb{G}^* \right\}$$

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

The sub-key for this protocol run will be:

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

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.

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

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.

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

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, which computes the corresponding public key and a proof of knowledge of S_j.

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 $\Gamma = \gamma_1, \ldots, \gamma_m$
- coefexps is set to the same value as the coefexps field of `vinputs`
- verification_keys is set to $\mathsf{public\_key}(\mathsf{vo}_1), \ldots, \mathsf{public\_key}(\mathsf{vo}_m)$

4.6 Trustees

$$\texttt{trustees} = \texttt{trustee\_kind}^*$$
$$\texttt{trustee\_kind} = [\texttt{"Single"}, \texttt{trustee\_public\_key}] \mid [\texttt{"Pedersen"}, \texttt{threshold\_parameters}]$$

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:

$$[[\texttt{"Single"}, \ldots], [\texttt{"Single"}, \ldots], [\texttt{"Single"}, \ldots]]$$

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:

$$[[\texttt{"Single"}, \ldots], [\texttt{"Pedersen"}, \ldots]]$$

As explained in section 3.1.1, the sub-keys of each item (`"Single"` or `"Pedersen"`) are then combined to form the global election key.

The server itself must always have a mandatory key, which must be different in each election. Other (third-party) keys may be imported from one election to another.

4.7 Credentials

A secret *credential* c is a string of the form `XXX-XXX-XXX-XXX-XXX` or `XXXXXXXXXXXXXXX`, where the 15 `X` 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` of the election as salt;
- 1000 iterations

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

4.8 Questions

$$\mathtt{question\_h} = \left\{ \begin{array}{rcl} \mathsf{answers} & : & \mathtt{string}^* \\ \mathsf{?blank} & : & \mathbb{B} \\ \mathsf{min} & : & \mathbb{I} \\ \mathsf{max} & : & \mathbb{I} \\ \mathsf{question} & : & \mathtt{string} \end{array} \right\} \qquad \begin{array}{rcl} \mathtt{question\_nh} & = & \left\{ \begin{array}{rcl} \mathsf{answers} & : & \mathtt{string}^* \\ \mathsf{question} & : & \mathtt{string} \end{array} \right\} \\ \mathtt{question\_gen} & = & \left\{ \begin{array}{rcl} \mathsf{type} & : & \mathtt{string} \\ \mathsf{value} & : & \mathtt{json} \end{array} \right\} \end{array}$$

$$\mathtt{question} = \mathtt{question\_h} \mid \mathtt{question\_gen}$$

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 Belenios. 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 `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

$$\mathtt{election} = \left\{ \begin{array}{rcl} \mathsf{version} & : & \mathbb{I} \\ \mathsf{description} & : & \mathtt{string} \\ \mathsf{name} & : & \mathtt{string} \\ \mathsf{group} & : & \mathtt{string} \\ \mathsf{public\_key} & : & \mathbb{G} \\ \mathsf{questions} & : & \mathtt{question}^* \\ \mathsf{uuid} & : & \mathtt{uuid} \\ \mathsf{?administrator} & : & \mathtt{string} \\ \mathsf{?credential\_authority} & : & \mathtt{string} \end{array} \right\}$$

The `election` structure includes all public data related to an election and is sent to each voter, serialized as a string which must be always the same throughout the election. The version is set to 1 in this version of the specification. It is incremented in case of backward-incompatible changes. The group is specified by the group member, either `BELENIOS-2048` or `RFC-3526-2048`. These groups are described in section 5, using the following structures:

$$\mathtt{embedding} = \left\{ \begin{array}{rcl} \mathsf{padding} & : & \mathbb{I} \\ \mathsf{bits\_per\_int} & : & \mathbb{I} \end{array} \right\} \qquad \mathtt{group} = \left\{ \begin{array}{rcl} \mathsf{g} & : & \mathbb{G} \\ \mathsf{p} & : & \mathbb{N} \\ \mathsf{q} & : & \mathbb{N} \\ \mathsf{?embedding} & : & \mathtt{embedding} \end{array} \right\}$$

The election public key, which is denoted by y thoughout this document, is computed during the setup phase, and stored in the $\mathsf{public\_key}$ member. 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 serialization of the `election` structure described above;
- the `trustees` structure described in section 4.6;
- the set L of public credentials.

Additionnally, we will denote throughout this document by φ the fingerprint of the election, as explained in section 4.14.

4.10 Encrypted answers

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

$$\texttt{answer\_nh} = \left\{ \begin{array}{rcl} \mathsf{choices} & : & \texttt{ciphertext} \\ \mathsf{proof} & : & \texttt{proof} \end{array} \right\}$$

$$\texttt{answer} = \texttt{answer\_h} \mid \texttt{answer\_nh}$$

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 be the number $\texttt{secret}(c)$, and S_0 be the string φ followed by a vertical bar and the serialization of g^s.

4.10.1 Homomorphic answers

An answer to a homomorphic question is the vector choices of encrypted values 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 value corresponding to whether the vote is blank or not. Each value 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 values respect constraints.

More concretely, each value $m \in [0 \dots 1]$ is encrypted (in an El Gamal-like fashion) into a `ciphertext` as follows:

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

where y is the election public key. The resulting vector is then used to compute S as follows:

1. let a be the vector choices, where each ciphertext c is replaced by the serialization of its alpha field, a comma, and the serialization of its beta field;
2. let b be the concatenation of all strings in a, separated by commas;
3. let S be the string S_0 followed by a vertical bar and b.

The individual proof that $m \in [0 \dots 1]$ is computed by running $\mathsf{iprove}(S_0, r, m, 0, 1)$ (see section 4.11).

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

When a blank vote is allowed, and there are n choices, the answer is modeled as a vector $(m_0, m_1, \dots, m_n)$, 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 $m_\Sigma = m_1 + \dots + m_n$. The additionnal proofs are as follows:

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

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 $[v_1, \dots, v_n]$ 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 κ and padding by p;
- it is assumed that each v_i is κ bits (or less);
- $[v_1, \dots, v_n]$ is encoded as:

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

  where ε (of p bits or less) is chosen so that $\xi \in \mathbb{G}$;
- choices is set to an El Gamal encryption of ξ as follows:
  1. pick a random $r \in \mathbb{Z}_q$
  2. $\mathsf{alpha} = g^r$
  3. $\mathsf{beta} = y^r \xi$

  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. $\mathsf{challenge} = \mathcal{H}_{\mathsf{raweg}}(S, y, \mathsf{alpha}, \mathsf{beta}, A)$
  4. $\mathsf{response} = w - r \times \mathsf{challenge}$

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

  $$\mathcal{H}_{\mathsf{raweg}}(S, y, \alpha, \beta, A) = \mathsf{SHA256}(\texttt{raweg|}S\texttt{|}y\texttt{,}\alpha\texttt{,}\beta\texttt{|}A) \mod q$$

  where `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 $A = g^{\mathsf{response}} \times \mathsf{alpha}^{\mathsf{challenge}}$
2. check that $\mathsf{challenge} = \mathcal{H}_{\mathsf{raweg}}(S, y, \mathsf{alpha}, \mathsf{beta}, A)$

4.11 Proofs of interval membership

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

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

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

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

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

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

The proof is verified as follows:

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

4.12 Proofs of possibly-blank votes

In this section, we suppose:

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

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

Additionnally, let $M_1, \ldots, M_k$ be the sequence $\mathsf{min}, \ldots, \mathsf{max}$ ($k = \mathsf{max} - \mathsf{min} + 1$).

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

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

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

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

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:

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

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

Computing overall_proof In $m_0 = 1 \vee m_\Sigma \in [M_1 \dots M_k]$, 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

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

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:

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

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

4.12.2 Blank votes ($m_0 = 1$)

Computing $\textsf{blank\_proof}$ In $m_0 = 0 \vee m_\Sigma = 0$, the second case is true. The proof $\textsf{blank\_proof}$ of the whole statement is the couple of proofs (π_0, π_Σ) built as in section 4.12.1, but exchanging subscripts 0 and Σ everywhere except in the call to $\mathcal{H}_{\textsf{bproof0}}$.

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

4.12.3 Verifying proofs

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

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

4.13 Signatures

$$\texttt{signature} = \left\{ \begin{array}{rcl} \mathsf{hash} & : & \texttt{string} \\ \mathsf{proof} & : & \texttt{proof} \end{array} \right\}$$

Each ballot contains a (Schnorr-like) digital signature to avoid ballot stuffing. The signature needs a credential c and uses the hash of the surrounding ballot (without the signature field). 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. compute proof as follows:
   (a) $\mathsf{challenge} = \mathcal{H}_{\mathsf{signature}}(\mathsf{hash}, A) \mod q$
   (b) $\mathsf{response} = w - s \times \mathsf{challenge} \mod q$

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

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

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 (credential and hash can be obtained from the surrounding ballot):

1. compute $A = g^{\mathsf{response}} \times \mathsf{credential}^{\mathsf{challenge}}$
2. check that $\mathsf{challenge} = \mathcal{H}_{\mathsf{signature}}(\mathsf{hash}, A) \mod q$

4.14 Ballots

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

A ballot references in its credential member the public credential $S = g^{\texttt{secret}(c)}$ (c being the secret credential) of the voter.

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

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

Where J is the serialization of the election structure, and the Base64 encoding is done without padding.

To compute the hash used in signatures, the ballot without the signature field is first serialized as a JSON compact string, where object fields are ordered as specified in this document. $\mathcal{H}_{\mathsf{JSON}}$ is then used on this serialization.

The same hashing function is used on the serialization of the whole ballot structure to produce a so-called *smart ballot tracker*.

The weight of a ballot B, denoted by $\mathsf{weight}(B)$, is the weight associated to $\mathsf{credential}(B)$ in the list of public credentials L.

4.15 Encrypted tally

$$\texttt{ciphertexts\_h} = \texttt{ciphertext}^* \qquad \texttt{ciphertexts\_nh} = \texttt{ciphertext}^*$$
$$\texttt{encrypted\_tally} = (\texttt{ciphertexts\_h} \mid \texttt{ciphertexts\_nh})^*$$

A so-called *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 (`ciphertexts_h`) is the pointwise product of the i-th ciphertext of all the ballots, raised to the power of its weight:

$$C_{i,j} = \prod_k \mathsf{choices}(\mathsf{answers}(B_k)_i)_j^{\mathsf{weight}(B_k)}$$

  where the product of two ciphertexts (α_1, β_1) and (α_2, β_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:

$$C_{i,k} = \mathsf{choices}(\mathsf{answers}(B_k)_i)$$

  In this case, it is an error if $\mathsf{weight}(B_k) \neq 1$.

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

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

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: if $i_1, \ldots, i_n$ are the indices of the non-homomorphic questions,

$$b = \mathsf{nh\_ciphertexts}(a) = [a_{i_1}, \ldots, a_{i_n}]$$

where a is the `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 $b' = \mathsf{nh\_ciphertexts}(a')$.

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 `shuffle` structure:

$$\texttt{shuffle} = \left\{ \begin{array}{rcl} \mathsf{ciphertexts} & : & \texttt{ciphertext}^{**} \\ \mathsf{proofs} & : & \texttt{shuffle\_proof}^* \end{array} \right\}$$

which uses the following auxiliary types:

$$\begin{array}{rcl} \texttt{shuffle\_commitment\_rand} & = & \mathbb{G} \times \mathbb{G} \times \mathbb{G} \times (\mathbb{G} \times \mathbb{G}) \times \mathbb{G}^* \\ \texttt{shuffle\_response} & = & \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^* \times \mathbb{Z}_q^* \\ \texttt{shuffle\_commitment\_perm} & = & \mathbb{G}^* \\ \texttt{shuffle\_chained\_challenges} & = & \mathbb{G}^* \\ \texttt{shuffle\_proof} & = & \texttt{shuffle\_commitment\_rand} \\ & \times & \texttt{shuffle\_response} \\ & \times & \texttt{shuffle\_commitment\_perm} \\ & \times & \texttt{shuffle\_chained\_challenges} \end{array}$$

For each non-homomorphic question i:

[2]See version 1.3.2 of the CHVote System Specification at [9]

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

The functions GenShuffle and GenShuffleProof are the same as in CHVote and are given in section 6. Typically, several shuffles will be computed sequentially by different persons.

4.17 Partial decryptions

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

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*:

$$\mathsf{decryption\_factors}_{i,j} = \mathsf{alpha}(a'_{i,j})^x$$

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

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

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

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

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
$$\begin{array}{rcl} A & = & g^{\mathsf{response}} \times \mathsf{public\_key}(k)^{\mathsf{challenge}} \\ B & = & \mathsf{alpha}(a'_{i,j})^{\mathsf{response}} \times \mathsf{decryption\_factors}_{i,j}^{\mathsf{challenge}} \end{array}$$
2. check that $\mathcal{H}_{\mathsf{decrypt}}(\mathsf{public\_key}(k), A, B) = \mathsf{challenge}$

4.18 Election result

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

The encrypted_tally field is set to the encrypted tally a'.

The decryption factors are combined for each ciphertext to build synthetic ones $F_{i,j}$. The way this combination is done depends on the `trustees` structure, the list PK. For each item of index τ in PK, a sub-factor $F_{i,j,\tau}$ is computed:

- for a "Single" item corresponding to trustee $\mathcal{T}_z$:

$$F_{i,j,\tau} = \mathsf{partial\_decryptions}_{z,i,j}$$

- for a "Pedersen" item corresponding to trustees $\mathcal{T}_{z_1}, \ldots, \mathcal{T}_{z_\mu}$:

$$F_{i,j,\tau} = \prod_{\delta \in \mathcal{I}} (\mathsf{partial\_decryptions}_{z_\delta,i,j})^{\lambda_\delta^{\mathcal{I}}}$$

  where $\mathcal{I}$ is the set of $(t+1)$ indexes of supplied partial decryptions, relative to $\mathcal{T}_{z_1}, \ldots, \mathcal{T}_{z_\mu}$ (i.e. $\mathcal{I} \subseteq \{1, \ldots, \mu\}$), and $\lambda_\delta^{\mathcal{I}}$ are the Lagrange coefficients:

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

The synthetic factor is then computed as the product of all sub-factors:

$$F_{i,j} = \prod_{\tau} F_{i,j,\tau}$$

The result field of the `result` structure is then computed as follows:

- if question i is homomorphic,

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

  where j represents an answer. The discrete logarithm can be easily computed because it is bounded by the sum of all weights;

- if question i is non-homomorphic,

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

  where j represents a ballot, and group_decode is the inverse of group_encode from section 4.10.2.

If the election has non-homomorphic questions, the shuffles field is set to the computed `shuffle` structures; otherwise, it is absent.

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 Group parameters

5.1 BELENIOS-2048

This group is optimized for elections that have only homomorphic questions and is used in this case. Its parameters have been generated by the `fips.sage` script (available in Belenios sources), which is itself based on FIPS 186-4.

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

The additional output of the generation algorithm is:

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

5.2 RFC-3526-2048

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:

$$
\begin{aligned}
\mathsf{p} &= \begin{array}{r}
32317006071311007 \\
300338913926423828248817941241140239112842009751400741706634 \\
354222619689417363569347117901737909704191754605873209195028 \\
853758986185622153212175412514901774520270235796078236248884 \\
246189477587641105928646099411723245426622522193230540919037 \\
680524235519125679715870117001058055877651038861847280257976 \\
054903569732561526167081339361799541336476559160368317896729 \\
073178384589680639671900977202194168647225871031411336429319 \\
536193471636533209717077448227988588565369208645296636077250 \\
268955505928362751121174096972998068410554359584866583291642 \\
136218231078990999448652468262416972035911852507045361090559
\end{array} \\
\mathsf{g} &= 2 \\
\mathsf{q} &= \begin{array}{r}
16158503035655503 \\
650169456963211914124408970620570119556421004875700370853317 \\
177111309844708681784673558950868954852095877302936604597514 \\
426879493092811076606087706257450887260135117898039118124442 \\
123094738793820552964323049705861622713311261096615270459518 \\
840262117759562839857935058500529027938825519430923640128988 \\
027451784866280763083540669680899770668238279580184158948364 \\
536589192294840319835950488601097084323612935515705668214659 \\
768096735818266604858538724113994294282684604322648318038625 \\
134477752964181375560587048486499034205277179792433291645821 \\
068109115539495499724326234131208486017955926253522680545279
\end{array}
\end{aligned}
$$

Additionally, its embedding field is set to:

$$
\left\{ \begin{array}{rcl} \mathsf{padding} & = & 8 \\ \mathsf{bits\_per\_int} & = & 8 \end{array} \right\}
$$

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 [9], 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

- $\mathbf{e} = [e_1, \ldots, e_N] \in \texttt{ciphertext}^N$: encrypted answers to one non-homomorphic question
- $y \in \mathbb{G}$: public key of the election

Algorithm

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

Table 1: Function $\mathsf{GenShuffle}(\mathbf{e}, y)$

Input

- $N \in \mathbb{N}$: permutation size

Algorithm

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

Table 2: Function $\mathsf{GenPermutation}(N)$

Input

- $e \in$ `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. $\alpha' \leftarrow \mathsf{alpha}(e) \times g^{r'}$
3. $\beta' \leftarrow \mathsf{beta}(e) \times y^{r'}$
4. Let e' be a new `ciphertext` with $\mathsf{alpha} = \alpha'$ and $\mathsf{beta} = \beta'$
5. Return (e', r') // $e' \in$ `ciphertext`, $r' \in \mathbb{Z}_q$

Table 3: Function $\mathsf{GenReEncryption}(e, y)$

Input

- $\mathbf{e} = [e_1, \ldots, e_N] \in \texttt{ciphertext}^N$: encrypted answers to one question; we will denote by α_i and β_i the contents of e_i
- $\mathbf{e}' = [e'_1, \ldots, e'_N] \in \texttt{ciphertext}^N$: shuffled encrypted answers; we will denote by α'_i and β'_i the contents of e'_i
- $\mathbf{r}' = [r'_1, \ldots, r'_N] \in \mathbb{Z}_q^N$: re-encryption randomizations
- $\psi = [j_1, \ldots, j_N] \in \Psi_N$: permutation
- $pk \in \mathbb{G}$: the public key of the election
- $\varphi \in \texttt{string}$: the fingerprint of the election

Algorithm

1. $h \leftarrow \mathsf{GetSecondaryGenerator}()$, $\mathbf{h} \leftarrow \mathsf{GetGenerators}(N)$ // see tables 6 and 7
2. $(\mathbf{c}, \mathbf{r}) \leftarrow \mathsf{GenPermutationCommitment}(\psi, \mathbf{h})$ // see table 9
3. $\mathsf{str}_c \leftarrow [\![\mathbf{e}]\!][\![\mathbf{e}']\!][\![\mathbf{c}]\!]$ // see table 10
4. $\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, \texttt{shuffle-challenges|}\varphi\texttt{|}\mathsf{str}_c)$ // see table 11
5. For $i = 1, \ldots, N$: $u'_i \leftarrow u_{j_i}$
6. $\mathbf{u}' \leftarrow [u'_1, \ldots, u'_N]$
7. $(\hat{\mathbf{c}}, \hat{\mathbf{r}}) \leftarrow \mathsf{GenCommitmentChain}(h, \mathbf{u}')$ // see table 12
8. For $i = 1, \ldots, 4$: pick ω_i at random in $\mathbb{Z}_q$
9. For $i = 1, \ldots, N$: pick $\hat{\omega}_i$ and ω'_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. $\hat{c}_0 \leftarrow h$
13. For $i = 1, \ldots, N$: $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \hat{c}_{i-1}^{\omega'_i}$
14. $t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\hat{t}_1, \ldots, \hat{t}_N])$, $\mathsf{str}_t \leftarrow [\![[t_1, t_2, t_3, t_{4,1}, t_{4,2}]]\!][\![[\hat{t}_1, \ldots, \hat{t}_N]]\!]$
15. $y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, $\mathsf{str}_y \leftarrow \mathsf{str}_c[\![\hat{\mathbf{c}}]\!]pk$ // pk taken as a number in base 10
16. $c \leftarrow \mathsf{GetNIZKPChallenge}(\texttt{shuffle-challenge|}\varphi\texttt{|}\mathsf{str}_t\mathsf{str}_y)$ // see table 13
17. $\bar{r} \leftarrow \sum_{i=1}^N r_i \mod q$, $s_1 \leftarrow \omega_1 + c \times \bar{r} \mod q$
18. $v_N \leftarrow 1$
19. For $i = N-1, \ldots, 1$: $v_i \leftarrow u'_{i+1} v_{i+1} \mod q$
20. $\hat{r} \leftarrow \sum_{i=1}^N \hat{r}_i v_i \mod q$, $s_2 \leftarrow \omega_2 + c \times \hat{r} \mod q$
21. $\tilde{r} \leftarrow \sum_{i=1}^N r_i u_i \mod q$, $s_3 \leftarrow \omega_3 + c \times \tilde{r} \mod q$
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 $i = 1, \ldots, N$: $\hat{s}_i \leftarrow \hat{\omega}_i + c \times \hat{r}_i \mod q$, $s'_i \leftarrow \omega'_i + c \times u'_i \mod q$
24. $s \leftarrow (s_1, s_2, s_3, s_4, [\hat{s}_1, \ldots, \hat{s}_N], [s'_1, \ldots, s'_N])$
25. $\pi \leftarrow (t, s, \mathbf{c}, \hat{\mathbf{c}})$
26. Return π // $\pi \in \texttt{shuffle\_proof}$

Table 4: Function $\mathsf{GenShuffleProof}(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi, pk, \varphi)$

Input

- $\pi \in$ `shuffle_proof`: shuffle proof
- $\mathbf{e} = [e_1, \dots, e_N] \in \texttt{ciphertext}^N$: encrypted answers to one question; we will denote by α_i and β_i the contents of e_i
- $\mathbf{e}' = [e'_1, \dots, e'_N] \in \texttt{ciphertext}^N$: shuffled encrypted answers; we will denote by α'_i and β'_i the contents of e'_i
- $pk \in \mathbb{G}$: the public key of the election
- $\varphi \in$ `string`: the fingerprint of the election

Algorithm

1. $(t, s, \mathbf{c}, \hat{\mathbf{c}}) \leftarrow \pi$
2. $(t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\hat{t}_1, \dots, \hat{t}_N]) \leftarrow t$
3. $(s_1, s_2, s_3, s_4, [\hat{s}_1, \dots, \hat{s}_N], [s'_1, \dots, s'_N]) \leftarrow s$
4. $[c_1, \dots, c_N] \leftarrow \mathbf{c}$, $[\hat{c}_1, \dots, \hat{c}_N] \leftarrow \hat{\mathbf{c}}$
5. $h \leftarrow \mathsf{GetSecondaryGenerator}()$, $\mathbf{h} \leftarrow \mathsf{GetGenerators}(N)$ // see tables 6 and 7
6. $\mathsf{str}_c \leftarrow [\![\mathbf{e}]\!][\![\mathbf{e}']\!][\![\mathbf{c}]\!]$ // see table 10
7. $\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, \texttt{shuffle-challenges|}\varphi\texttt{|}\mathsf{str}_c)$ // see table 11
8. $\mathsf{str}_t \leftarrow [\![[t_1, t_2, t_3, t_{4,1}, t_{4,2}]]\!][\![[\hat{t}_1, \dots, \hat{t}_N]]\!]$
9. $\mathsf{str}_y \leftarrow \mathsf{str}_c[\![\hat{\mathbf{c}}]\!]pk$ // pk taken as a number in base 10
10. $c \leftarrow \mathsf{GetNIZKPChallenge}(\texttt{shuffle-challenge|}\varphi\texttt{|}\mathsf{str}_t\mathsf{str}_y)$ // see table 13
11. $\bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i$
12. $u \leftarrow \prod_{i=1}^N u_i \mod q$
13. $\hat{c}_0 \leftarrow h$
14. $\hat{c} \leftarrow \hat{c}_N / h^u$
15. $\tilde{c} \leftarrow \prod_{i=1}^N c_i^{u_i}$
16. $(\alpha', \beta') \leftarrow (\prod_{i=1}^N \alpha_i^{u_i}, \prod_{i=1}^N \beta_i^{u_i})$
17. $t'_1 \leftarrow \bar{c}^{-c} \times g^{s_1}$
18. $t'_2 \leftarrow \hat{c}^{-c} \times g^{s_2}$
19. $t'_3 \leftarrow \tilde{c}^{-c} \times g^{s_3} \prod_{i=1}^N h_i^{s'_i}$
20. $(t'_{4,1}, t'_{4,2}) \leftarrow ((\beta')^{-c} \times pk^{-s_4} \prod_{i=1}^N (\beta'_i)^{s'_i}, (\alpha')^{-c} \times g^{-s_4} \prod_{i=1}^N (\alpha'_i)^{s'_i})$
21. For $i = 1, \dots, N$: $\hat{t}'_i \leftarrow \hat{c}_i^{-c} \times g^{\hat{s}_i} \times \hat{c}_{i-1}^{s'_i}$
22. Return $(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3) \wedge (t_{4,1} = t'_{4,1}) \wedge (t_{4,2} = t'_{4,2}) \wedge \left[\bigwedge_{i=1}^N (\hat{t}_i = \hat{t}'_i)\right]$

Table 5: Function $\mathsf{CheckShuffleProof}(\pi, \mathbf{e}, \mathbf{e}', pk, \varphi)$

Algorithm

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

Table 6: Function $\mathsf{GetSecondaryGenerator}()$

Input

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

Algorithm

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

Table 7: Function $\mathsf{GetGenerators}(N)$

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. $c \leftarrow (p-1)/q$ // typically, $c = 2$
2. $x \leftarrow \mathsf{SHA256}(\texttt{ggen}|i)$ // 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 // $h \in \mathbb{G}$

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

Input

- $\psi = [j_1, \ldots, j_N] \in \Psi_N$: permutation
- $\mathbf{h} = [h_1, \ldots, h_N] \in \mathbb{G}^N$: 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. $\mathbf{c} \leftarrow [c_1, \ldots, c_N]$
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 9: Function $\mathsf{GenPermutationCommitment}(\psi, \mathbf{h})$

Input

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

Algorithm

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

Table 10: Functions $[\![\mathbf{e}]\!]$ and $[\![\mathbf{c}]\!]$

Input

- $N \in \mathbb{N}$: number of ciphertexts
- $S \in \texttt{string}$: challenge string

Algorithm

1. $H \leftarrow \mathsf{SHA256}(S)$ // output interpreted as an hexadecimal string
2. For $i = 0, \ldots, N-1$:
   (a) $T \leftarrow \mathsf{SHA256}(i)$ // input taken as decimal, output interpreted as hexadecimal
   (b) $u_i \leftarrow \mathsf{SHA256}(HT) \mod q$ // 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 $\mathsf{GetNIZPKChallenges}(N, S)$

Input

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

Algorithm

1. For $i = 1, \ldots, N$:
   (a) Pick r_i at random in $\mathbb{Z}_q$
   (b) $c_i \leftarrow g^{r_i} \times c_{i-1}^{u_i}$
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 $\mathsf{GenCommitmentChain}(c_0, \mathbf{u})$

Input

- $S \in$ `string`: challenge string

Algorithm

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

Table 13: Function $\mathsf{GetNIZPKChallenge}(S)$

References

[1] B. Adida. Helios: web-based open-audit voting. In *Proceedings of the 17th conference on Security symposium (SS 2018)*, SS'08, pages 335–348, Berkeley, CA, USA. USENIX Association.

[2] D. Bernhard, B. Warinschi, and O. Pereira. How not to prove yourself: Pitfalls of Fiat-Shamir and applications to Helios. In *Advances in Cryptology (AsiaCrypt 2012)*, volume 7658 of *LNCS*, pages 626–643, 2012.

[3] V. Cortier, C. C. Dragan, P.-Y. Strub, F. Dupressoir, and B. Warinschi. Machine-checked proofs for electronic voting: privacy and verifiability for Belenios. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF 2018)*, pages 298–312, 2018.

[4] V. Cortier, D. Galindo, S. Glondu, and M. Izabachene. Distributed ElGamal à la Pedersen - Application to Helios. In *Workshop on Privacy in the Electronic Society (WPES 2013)*, Berlin, Germany, 2013.

[5] V. Cortier, D. Galindo, S. Glondu, and M. Izabachene. Election verifiability for Helios under weaker trust assumptions. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS 2014)*, volume 8713 of *LNCS*, pages 327–344, Wroclaw, Poland, 2014. Springer.

[6] V. Cortier, P. Gaudry, and S. Glondu. Belenios: A simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, pages 214–238. Springer International Publishing, 2019.

[7] V. Cortier, P. Gaudry, and Q. Yang. How to fake zero-knowledge proofs, again. In *Fifth International Joint Conference on Electronic Voting (E-Vote-ID 2020)*, Bregenz / virtual, Austria, 2020.

[8] P. Gaudry. Some ZK security proofs for Belenios. working paper or preprint, 2017.

[9] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote system specification. Cryptology ePrint Archive, Report 2017/325, 2017. `https://eprint.iacr.org/2017/325`.

[10] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1991*, pages 129—140, 1991.