Belenios specification

Stéphane Glondu

Version 2.3

Contents

1 Introduction

References. This document is a specification of the voting protocol implemented in Belenios 2.3 . A high level description of Belenios and some statistics about its usage can be found in [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. In practice, $\mathbb{G}$ can be either a prime order multiplicative subgroup of $\mathbb{F}_p^*$ (hence, all exponentiations are implicitly done modulo p), or a prime order subgroup of the points of an elliptic curve. We suppose the group parameters are agreed on beforehand. Examples of supported groups are given 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
- $\mathcal{T}_1, \ldots, \mathcal{T}_m$: trustees
- $\mathcal{V}_1, \ldots, \mathcal{V}_n$: 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 (see 4.4) that consists of a sequence of data and events, and is structured as a hash chain. It contains in particular:
  - 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)

  The voting server also produces a list PB of pretty ballots, that is a list of hashes of ballots (the last ballot of each voter).

3 Processes

3.1 Election setup

1. $\mathcal{A}$ starts the preparation of an election, providing in particular the questions and the list of voters
2. $\mathcal{S}$ generates a fresh `uuid` u and sends it to $\mathcal{C}$
3. $\mathcal{C}$ generates random private credentials and salts $c_1, \ldots, c_n$ and $s_1, \ldots, s_n$ and computes

   $$L = \mathsf{sort}((\mathsf{public}(c_1, s_1), w_1, \mathcal{V}_1, s_1), \ldots, (\mathsf{public}(c_n, s_n), w_n, \mathcal{V}_n, s_n))$$

   (for each c_i, the index suffix in c_i is set to its index in L)
4. for $j \in [1 \ldots n]$, $\mathcal{C}$ sends c_j to $\mathcal{V}_j$
5. (optional) $\mathcal{C}$ forgets $c_1, \ldots, c_n$
6. $\mathcal{C}$ sends L to $\mathcal{S}$
7. $\mathcal{S}$ checks that the $\mathcal{V}_i$ and the w_i in L are correct, and that all public credentials and salts are distinct, and extract the salt list as `salts`;

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 $\mathcal{T}_1, \ldots, \mathcal{T}_m$ run key establishment protocols (see 3.1.1) as needed to fill in the `trustees` structure;
10. $\mathcal{S}$ creates the `election` E
11. $\mathcal{S}$ publishes `salts`
12. $\mathcal{C}$ checks that the list of public credentials in L is exactly the one that appears on the public data of the election.

Step 5 is optional. It offers a better protection against ballot stuffing in case $\mathcal{C}$ unintentionally leaks private credentials (but disables credential recovery, see Section 3.3).

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_τ. 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:

$$y = \prod_\tau y_\tau$$

`"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 4.6.4.

3.2 Vote

1. $\mathcal{V}$ gets public data of E
2. $\mathcal{V}$ uses the index in her secret credential c to get her public credential $\hat{c}$ (from election public data) and her salt s (from `salts`), and checks that $\hat{c} = \mathsf{public}(c, s)$
3. $\mathcal{V}$ creates a `ballot` b, she computes the hash h of b, called tracking number, and submits b to $\mathcal{S}$
4. $\mathcal{S}$ processes b:
   (a) let C be the public credential used in b (its $\mathsf{credential}$ field)
   (b) $\mathcal{S}$ checks that $(C, w_i, \mathcal{V}) \in L$
   (c) $\mathcal{S}$ checks all zero-knowledge proofs of b
   (d) $\mathcal{S}$ adds b to D
5. at any time (even after tally), $\mathcal{V}$ may check that h (if it is her last ballot) appears in the list of pretty ballots PB and the weight of her ballot as it appears in PB 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 to $\mathcal{V}_i$

3.4 Tally

1. $\mathcal{A}$ stops $\mathcal{S}$, $\mathcal{S}$ computes the initial `encrypted_tally` Π_0, and publishes it in D
2. $\mathcal{S}$ extracts the non-homomorphic ciphertexts from the encrypted tally (see section 4.17):

$$\tilde{\Pi}_0 = \mathsf{nh\_ciphertexts}(\Pi_0)$$

3. if the election contains a non-homomorphic part, that is, if $\tilde{\Pi}_0 \neq []$, then for $z \in [1 \ldots m]$:

(a) $\mathcal{S}$ sends $\tilde{\Pi}_{z-1}$ to $\mathcal{T}_z$

(b) $\mathcal{T}_z$ verifies consistency of D

(c) $\mathcal{T}_z$ runs the shuffle algorithm, producing a `shuffle` σ_z and sends it to $\mathcal{S}$

(d) $\mathcal{S}$ verifies σ_z, publishes it in D, 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) $\mathcal{T}_z$ verifies consistency of D

(c) $\mathcal{T}_z$ generates a `partial_decryption` δ_z and sends it to $\mathcal{S}$

(d) $\mathcal{S}$ verifies δ_z and adds it to D

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 (if any) appear 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 gets the public data D and retrieves the list L of public credentials;

   - she records D;
   - for each ballot b in D, 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 computes $\hat{B} = \mathsf{last}(B)$, the list of ballots obtained from B by removing all ballots that have the same credential except the last one (only the last vote is kept for each voter, see section 4.15). She checks that the list of hashed ballots in $\hat{B}$ corresponds to the pretty ballots PB;
   - she checks that D is correctly chained, that is, each event correctly refers to the hash of its parent's event.

2. she retrieves the previously recorded election data D' (if it exists) and gets the hash h of the last event in D'. She checks that h appears as the hash of a parent of an event in D. This ensures that nothing has been removed from D'.

There is no tool support on the web interface for these checks, instead the command line tools `election verify` and `election verify-diff` can be used.

3.5.2 After the tally

The auditor retrieves the public data D and in particular the list B of ballots, the list Σ of shuffles (if any), the list Δ of partial decryptions 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 computes $\hat{B} = \mathsf{last}(B)$, that is, she keeps only the last ballots (see section 4.15);
4. she checks that the proofs in Σ and Δ, and the result r, are valid w.r.t. $\hat{B}$.

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 `election verify` can be used.

4 Messages

4.1 Conventions

Structured data is encoded in JSON (RFC 4627). When serialized, data must be in compact form, and order of fields must be respected. 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
- $\mathbb{N}$, $\mathbb{Z}_q$: big integer, written in base 10 and encoded as a JSON string
- $\mathbb{G}$: a JSON string, whose interpretation depends on the group
- $\mathbb{H}$: hash (typically SHA256), written in hexadecimal and encoded as a JSON string

4.3 Common structures

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

[1] Base58 characters are: `123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

4.4 Public data

During an election, all public data is published in a dynamic file. This file can be used to perform verifications and monitoring. It is actually an old-style tar archive and evolves in an append-only fashion. It stops evolving when the election is tallied.

The archive starts with a `BELENIOS` file containing a JSON structure that acts as a header for the whole archive. It contains a version number and the timestamp of the beginning of the election. It is then followed by JSON files of two kinds: data (whose names end in `.data.json`) and events (whose names end in `.event.json`). The file names start with the SHA256 hash of their contents, encoded in hexadecimal.

$$\texttt{header} = \left\{ \begin{array}{rcl} \mathsf{version} & : & \mathbb{I} \\ \mathsf{timestamp} & : & \mathbb{I} \end{array} \right\} \qquad \texttt{event} = \left\{ \begin{array}{rcl} ?\mathsf{parent} & : & \mathbb{H} \\ \mathsf{height} & : & \mathbb{I} \\ \mathsf{type} & : & \texttt{event\_type} \\ ?\mathsf{payload} & : & \mathbb{H} \end{array} \right\}$$

$$\begin{array}{rl} & \texttt{"Setup"} \\ & \mid \texttt{"Ballot"} \mid \texttt{"EndBallots"} \\ \texttt{event\_type} = & \mid \texttt{"EncryptedTally"} \\ & \mid \texttt{"Shuffle"} \mid \texttt{"EndShuffles"} \\ & \mid \texttt{"PartialDecryption"} \mid \texttt{"Result"} \end{array}$$

An `event` structure is appended at each important step of the election (e.g. when somebody votes). This structure refers to its predecessor (except for the first one) through its parent field, and can refer to a payload by its hash. Its height field is an integer, set to 0 in the first event, and incremented in each new event. Typically, the payload will precede the `event` structure in the archive, and can itself refer to other payloads that precede. The type of the payload depends on the type field of the `event` structure.

The typical sequence of data and events occurring in an archive is described in the following table:

Data	Event	Defined in section
`election`		4.10
`trustees`		4.7
`public_credentials`		4.8
`setup_data`		4.10
	`Setup`	
`ballot`		4.15
	`Ballot`	
⋮	⋮	
	`EndBallots`	
`encrypted_tally`		4.16
`sized_encrypted_tally`		4.16
	`EncryptedTally`	
`shuffle`		4.17
`owned_shuffle`		4.17
	`Shuffle`	
⋮	⋮	
	`EndShuffles`	
`partial_decryption`		4.18
`owned_partial_decryption`		4.18
	`PartialDecryption`	
⋮	⋮	
`result`		4.19
	`Result`	

4.5 Verification keys

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

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

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

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

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

where pok and the vertical bars are verbatim, 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 $A = g^{\mathsf{response}} \times X^{\mathsf{challenge}}$
2. check that $\mathsf{challenge} = \mathcal{H}_{\mathsf{pok}}(X, A) \mod q$

4.6 Messages specific to threshold decryption support

4.6.1 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
```

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.

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

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 $\mathsf{SHA256}(\texttt{sigmsg|}\, M \,|\, A)$, where 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 $\mathsf{challenge} = \mathsf{SHA256}(\texttt{sigmsg|}\, M \,|\, A)$

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.

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

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 $\mathsf{SHA256}(\texttt{key} \mid g^s)$ as the key and $\mathsf{SHA256}(\texttt{iv} \mid g^r)$ as the initialization vector

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

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

4.6.2 Certificates

A certificate is a `signed_msg` encapsulating a serialized `cert_keys` structure, itself filled with the public keys generated as described in section 4.6.1.

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

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

4.6.3 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 (typically the election server).

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

When decoding such a message, recipient must be checked.

4.6.4 Polynomials

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

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

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

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

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

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

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

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

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

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

  filled with s_{ij}

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

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

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

The sub-key for this protocol run will be:

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

4.6.5 Vinputs

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

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

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

- the 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 coefexps field is $\mathsf{coefexps}(P_1), \ldots, \mathsf{coefexps}(P_m)$

Note that the 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 $i = 1, \ldots, m$,

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

4.6.6 Voutputs

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

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

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

- 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:

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

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

4.6.7 Threshold parameters

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

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

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 $\mathsf{public\_key}(\mathsf{vo}_1), \ldots, \mathsf{public\_key}(\mathsf{vo}_m)$

4.7 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 4.5, or threshold parameters as described in section 4.6. 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.

4.8 Credentials

A secret *credential* c is a string of the form `XXX-XXXX-XXX-XXXX-NNNN`, where the 14 `X` characters are taken from the Base58 alphabet:

`123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

The string `XXXXXXXXXXXXXX` will be called the *prefix* of c. `NNNN` is a base 10 number that serves as an index to locate the salt associated to c in the list of salts and the matching public key in the election public data. A *salt* s is a string of 22 characters from the Base58 alphabet.

From c and s, a secret exponent $x = \mathsf{secret}(c, s)$ is derived by using PBKDF2 (RFC 2898) with:

- the prefix of c as password;
- HMAC-SHA256 (RFC 2104, FIPS PUB 180-2) as pseudorandom function;
- the concatenation of the `uuid` of the election and s as salt;
- 100 000 iterations

and an output size of 2 blocks, which is interpreted as a big-endian 512-bit number and then reduced modulo q to form x. From this secret exponent, a public key $\mathsf{public}(c, s) = g^x$ is computed.

$$\texttt{public\_credentials} = \texttt{string}^*$$

Public credentials, published as part of public data, are a list of strings, each one being:

- either a public credential by itself (if weights are not being used),
- or a public credential, followed by a comma and the weight (if weights are being used).

4.9 Questions

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

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

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 4.11) will be revealed at the end of the election whereas with a non-homomorphic question, each individual answer will be revealed.

4.9.1 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).

4.9.2 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. An extra field may be present, to give a hint on the intended counting method (Majority Judgment, Condorcet, STV, ...).

4.10 Elections

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

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, a short string unambiguously describing the group (see section 5).

The election public key, which is denoted by y throughout this document, is computed during the setup phase, and stored in the public_key member.

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 4.7;
- the `public_credentials` structure described in section 4.8.

These three structures are referred to by the following structure, used as payload of the `Setup` event in public data.

$$\texttt{setup\_data} = \left\{ \begin{array}{rcl} \text{election} & : & \mathbb{H} \\ \text{trustees} & : & \mathbb{H} \\ \text{credentials} & : & \mathbb{H} \end{array} \right\}$$

Additionally, we will denote throughout this document by φ the Base64 encoding of the election field of `setup_data`, without padding.

4.11 Encrypted answers

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

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

`answer` = `answer_h` | `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 $\mathsf{secret}(c)$, and S_0 be the string φ followed by a vertical bar and the serialization of g^s.

4.11.1 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. alpha $= g^r$
3. 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 4.12).

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 additional 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 4.13.

4.11.2 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, $\mathbb{G}$ must support the to_ints and of_ints operations (see section 5). The answer is encrypted as follows:

- $\xi = \mathsf{of\_ints}([v_1, \dots, v_n])$
- 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:

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

  where `raweg`, vertical bars and commas are verbatim. 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 $\mathsf{challenge} = \mathcal{H}_{\mathsf{raweg}}(S, y, \mathsf{alpha}, \mathsf{beta}, A)$

4.12 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 challenge and a random 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:

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

where `prove`, vertical bars and commas are verbatim. 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

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

4.13 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.

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

4.13.1 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:

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

where `bproof0`, vertical bars and commas are verbatim. The result is interpreted as a 256-bit big-endian number.

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

1. pick random $\mathsf{challenge}(\pi_0)$ and $\mathsf{response}(\pi_0)$ in $\mathbb{Z}_q$
2. compute $A_0 = g^{\mathsf{response}(\pi_0)} \times \alpha_0^{\mathsf{challenge}(\pi_0)}$ and $B_0 = y^{\mathsf{response}(\pi_0)} \times (\beta_0/g)^{\mathsf{challenge}(\pi_0)}$
3. for $j > 0$ and $j \neq i$:
   (a) create π_j with a random challenge and a random response in $\mathbb{Z}_q$
   (b) compute $A_j = g^{\mathsf{response}} \times \alpha_\Sigma^{\mathsf{challenge}}$ and $B_j = y^{\mathsf{response}} \times (\beta_\Sigma / g^{M_j})^{\mathsf{challenge}}$
4. pick a random $w \in \mathbb{Z}_q$
5. compute $A_i = g^w$ and $B_i = y^w$
6. compute
$$\mathsf{challenge}(\pi_i) = \mathcal{H}_{\mathsf{bproof1}}(S, A_0, B_0, \ldots, A_k, B_k) - \sum_{j \neq i} \mathsf{challenge}(\pi_j) \mod q$$
7. compute $\mathsf{response}(\pi_i) = w - r_\Sigma \times \mathsf{challenge}(\pi_i) \mod q$

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

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

where `bproof1`, vertical bars and commas are verbatim. The result is interpreted as a 256-bit big-endian number.

4.13.2 Blank votes ($m_0 = 1$)

Computing blank_proof In $m_0 = 0 \vee m_\Sigma = 0$, the second case is true. The proof blank_proof of the whole statement is the couple of proofs (π_0, π_Σ) built as in section 4.13.1, but exchanging subscripts 0 and Σ everywhere except in the call to $\mathcal{H}_{\mathsf{bproof0}}$.

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

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

4.13.3 Verifying proofs

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

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

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

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

4.14 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 $s = \mathsf{secret}(c)$
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`, vertical bars and commas are verbatim. 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 $\mathsf{challenge} = \mathcal{H}_{\mathsf{signature}}(\mathsf{hash}, A) \mod q$

4.15 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 $S = g^{\mathsf{secret}(c,s)}$ (c being the secret credential, s being the associated salt) of the voter. The hash (or *fingerprint*) of the election is φ (see section 4.10).

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. The hash is the compact Base64 encoding of the SHA256 of this string. 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.

Ballots are appended to public data as payloads of `Ballot` events. The end of ballots is marked by an `EndBallots` event without payload.

Tallied ballots The list of tallied ballots $\hat{B}$ can be computed from the list of all accepted ballots B with the last function ($\hat{B} = \mathsf{last}(B)$), defined as follows:

- initialize $\hat{B}$ with the empty list
- for $b \in B$:
  - if there is a ballot in $\hat{B}$ with the same credential as b, remove it
  - add b to $\hat{B}$

4.16 Encrypted tally

$$\texttt{ciphertexts\_h} = \texttt{ciphertext}^* \qquad \texttt{ciphertexts\_nh} = \texttt{ciphertext}^*$$

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

$$\texttt{sized\_encrypted\_tally} = \left\{ \begin{array}{rcl} \mathsf{num\_tallied} & : & \mathbb{I} \\ \mathsf{total\_weight} & : & \mathbb{I} \\ \mathsf{encrypted\_tally} & : & \mathbb{H} \end{array} \right\}$$

A so-called *encrypted tally* is constructed out of the accepted ballots $\hat{B} = \mathsf{last}(B) = b_1, \ldots, b_n$ (see section 4.15). 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:

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

The `sized_encrypted_tally` structure contains the number of ballots taken into account, their total weight, and a reference to the `encrypted_tally` structure. It is used as the payload of the `EncryptedTally` event.

4.17 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 4.16. 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:

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.

$$\texttt{owned\_shuffle} = \left\{ \begin{array}{rcl} \mathsf{owner} & : & \mathbb{I} \\ \mathsf{payload} & : & \mathbb{H} \end{array} \right\}$$

The `owned_shuffle` structure links a shuffle with the trustee that did it and is used as payload of `Shuffle` events.

4.18 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. $\mathsf{challenge} = \mathcal{H}_{\mathsf{decrypt}}(X, A, B)$

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

4. $\mathsf{response} = w - x \times \mathsf{challenge} \mod q$

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

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

where `decrypt`, vertical bars and commas are verbatim. 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
$$\begin{aligned} A &= g^{\mathsf{response}} \times \mathsf{public\_key}(k)^{\mathsf{challenge}} \\ B &= \mathsf{alpha}(a'_{i,j})^{\mathsf{response}} \times \mathsf{decryption\_factors}_{i,j}^{\mathsf{challenge}} \end{aligned}$$
2. check that $\mathcal{H}_{\mathsf{decrypt}}(\mathsf{public\_key}(k), A, B) = \mathsf{challenge}$

$$\texttt{owned\_partial\_decryption} = \left\{ \begin{array}{rcl} \mathsf{owner} & : & \mathbb{I} \\ \mathsf{payload} & : & \mathbb{H} \end{array} \right\}$$

The `owned_partial_decryption` structure links a partial decryption with the trustee that did it and is used as payload of `PartialDecryption` events.

4.19 Election result

$$\texttt{result} = \left\{ \begin{array}{rcl} \mathsf{result} & : & (\mathbb{I}^* \mid \mathbb{I}^{**})^* \end{array} \right\}$$

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,

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

  where j represents a ballot.

5 Groups

A group is identified by a short string. In addition to the usual mathematical group definition, it must support the following operations:

- check: checking that an element of the surrounding set is indeed a group element;
- to_string, of_string: (de-)serialization to/from strings, used when serializing JSON structures or computing hashes.

Additionally, it may support the following operations, needed for non-homomorphic questions (see section 4.11.2):

- to_ints, of_ints: embedding of vectors of small integers into group elements,
- get_generator: a function mapping integers to group generators different from g.

Supported groups include:

- BELENIOS-2048
- RFC-3526-2048
- Ed25519

5.1 Finite fields

Here, groups are multiplicative subgroups of $\mathbb{F}_p^*$ described by the following structure:

$$\texttt{group} = \left\{ \begin{array}{rcl} \mathsf{g} & : & \mathbb{G} \\ \mathsf{p} & : & \mathbb{N} \\ \mathsf{q} & : & \mathbb{N} \\ \mathsf{?embedding} & : & \texttt{embedding} \end{array} \right\}$$

When serialized as strings, group elements are written in base 10.

These groups may support non-homomorphic encoding described by the following structure:

$$\texttt{embedding} = \left\{ \begin{array}{rcl} \mathsf{padding} & : & \mathbb{I} \\ \mathsf{bits\_per\_int} & : & \mathbb{I} \end{array} \right\}$$

The encoding works as follows:

- 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);

- $[v_1, \ldots, v_n]$ is encoded as:

$$\xi = \mathsf{of\_ints}([v_1, \ldots, v_n]) = (((v_1 \times 2^\kappa + v_2) \times 2^\kappa + \cdots) \times 2^\kappa + v_n) \times 2^p + \varepsilon$$

  where ε (of p bits or less) is chosen so that $\xi \in \mathbb{G}$;

- the to_ints is the inverse of of_ints, and takes as input a group element ξ and the number n of encoded integers.

The get_generator function is the pure part of GetGenerator defined in table 8.

5.1.1 BELENIOS-2048

This group is optimized for elections that have only homomorphic questions. It has no embedding. Its parameters have been generated by the `fips.sage` script (available in Belenios sources), which is itself based on FIPS 186-4.

```
p = 20694785691422546
    40101364365750500806492298929575110409710088478705737421924 2
    71740192223725449768433812906663313807895840496005438963628 9
    79639303877390572280360597374942767137677761889858987273586 5
    04908116709931053586778098003079049165406377717376419867852 7
    27347447634183560003569830519314428456170191100078673730733 3
    56412397173289791324047457883446826065232797464795113767265 8
    69358218004631792207366886005262718636338608879688212076943 2
    36614949100292344434637322145884100586421050242120365433561
    20132048111885240873107701415166620016231317716937218924807 8
    50771182784231749807327659882882516918310312568016207288071 9
g = 2402352677501852
    20922768770353239993271228765737836491651007531878766327414 6
    35321932028567615526967879969466829874938909508389657342560 1
    90060106847716449173547413728310461045868131451178164675540 0
    52740288984613986453266121505579709716201616827031288643245 6
    66383486363578210615491841998253431518974065818686865115135 8
    57641013888221539601604322884360393098933366272848406593138
    40601023167509576377798266510360682240663507669776402534625 3
    77308513317349519424896775405257365904949247763147599157519 8
    77517771148149092045660020547812705472823814097251863985833 4
    11570056835369555342378147558249189605029668003774530846062 7
q = 78571733251071885
    07992765981267145012182142125840879461151008191980562322344 1
```

The additional output of the generation algorithm is:

```
domain_parameter_seed = 478953892617249466
                        16610647609884762656313816802 7
                        71688248873244719834900039659 2
                        02063287517272455214556016774 6
counter = 109
```

5.1.2 RFC-3526-2048

The group described in the previous section is not suitable for encoding non-homomorphic answers. Therefore, we describe here a different group for cases where the election has non-homomorphic

questions. This group is the 2048-bit one defined in RFC 3526:

```
p = 32317006071311007
    30033891392642382824881794124114023911284200975140074170663
    35422261968941736356934711790173790970419175460587320919502
    85375898618562215321217541251490177452027023579607823624888
    24618947758764110592864609941172324542662252219323054091903
    68052423551912567971587011700105805587765103886184728025797
    05490356973256152616708133936179954133647655916036831789672
    07317838458968063967190097720219416864722587103141133642931
    53619347163653320971707744822798858856536920864529663607725
    26895550592836275112117409697299806841055435958486658329164
    13621823107899099944865246826241697203591185250704536109055
g = 2
q = 16158503035655503
    65016945696321191412440897062057011955642100487570037085331
    77111309844708681784673558950868954852095877302936604597514
    42687949309281107660608770625745088726013511789803911812444
    12309473879382055296432304970586162271331126109661527045951
    84026211775956283985793505850052902793882551943092364012898
    02745178486628076308354066968089977066823827958018415894836
    53658919229484031983595048860109708432361293551570566821465
    76809673581826660485853872411399429428268460432264831803862
    13447775296418137556058704848649903420527717979243329164582
    06810911553949549972432623413120848601795592625352268054527
```

Additionally, its embedding field is set to:

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

5.2 Elliptic curves

5.2.1 Ed25519

The Ed25519 group is a well-known group, defined in RFC 7748. It is defined by the following parameters:

- $p = 2^{255} - 19$,
- $E/\mathbb{F}_p$ is the twisted Edwards curve:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2,$$

- g is the unique point in $E(\mathbb{F}_p)$ whose y coordinate is $4/5$ and whose x coordinate is positive,
- the order of g is $q = 2^{252} + 27742317777372353535851937790883648493$.

In the above, *positive* is defined in terms of bit-encoding:

- *positive* coordinates are even coordinates (least significant bit is cleared),
- *negative* coordinates are odd coordinates (least significant bit is set).

Group elements are points represented by their coordinates (x, y), which can be *compressed* into a single 256-bit number $z = \mathsf{compress}(x, y)$:

- let b be the least significant bit of x shifted to the left by 255 bits,

- let z be the binary XOR of b and y.

This z can be uncompressed into a single point $\mathsf{uncompress}(z)$. This compressed form is used for serialization, written in hexadecimal, always padded with leading-zeroes so that the resulting string is always 64 characters.

Additionally, a non-homomorphic encoding is defined as follows:

- let $\kappa = 8$ and $p = 14$;
- it is assumed that each small integer is κ bits (or less);
- $[v_1, \ldots, v_n]$ is encoded as:

  $$\xi = \mathsf{of\_ints}([v_1, \ldots, v_n]) = \mathsf{uncompress}((((v_1 \times 2^\kappa + v_2) \times 2^\kappa + \cdots) \times 2^\kappa + v_n) \times 2^p + \varepsilon)$$

  where ε (of p bits or less) is the smallest non-negative integer such that $\xi \in \mathbb{G}$;
- the $\mathsf{to\_ints}$ is the inverse of $\mathsf{of\_ints}$, and takes as input a group element ξ and the number n of encoded integers.

The $\mathsf{get\_generator}$ function is the pure part of $\mathsf{GetGenerator}$ defined in table 9.

6 Shuffle algorithms

The algorithms $\mathsf{GenShuffle}$ and $\mathsf{GenShuffleProof}$ are referred to in section 4.17. 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 $\mathsf{CheckShuffleProof}$ algorithm, used to check a proof produced by $\mathsf{GenShuffleProof}$. 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

- $e \in \texttt{ciphertext}$: 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') // $e' \in \texttt{ciphertext}$, $r' \in \mathbb{Z}_q$

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 10
3. $\mathsf{str}_c \leftarrow [\![\mathbf{e}]\!][\![\mathbf{e}']\!][\![\mathbf{c}]\!]$ // see table 11
4. $\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, \texttt{shuffle-challenges} \,|\, \varphi \,|\, \mathsf{str}_c)$ // see table 12
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 13
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. $y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, $\mathsf{str}_y \leftarrow \mathsf{str}_c [\![\hat{\mathbf{c}}]\!] pk$
16. $c \leftarrow \mathsf{GetNIZKPChallenge}(\texttt{shuffle-challenge} \,|\, \varphi \,|\, \mathsf{str}_t \mathsf{str}_y)$ // see table 14
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
- $\mathbf{e} = [e_1, \ldots, e_N] \in \texttt{ciphertext}^N$: 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
- $pk \in \mathbb{G}$: 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. $(t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\hat{t}_1, \ldots, \hat{t}_N]) \leftarrow t$
3. $(s_1, s_2, s_3, s_4, [\hat{s}_1, \ldots, \hat{s}_N], [s'_1, \ldots, s'_N]) \leftarrow s$
4. $[c_1, \ldots, c_N] \leftarrow \mathbf{c}$, $[\hat{c}_1, \ldots, \hat{c}_N] \leftarrow \hat{\mathbf{c}}$
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 11
7. $\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, \texttt{shuffle-challenges} \,|\, \varphi \,|\, \mathsf{str}_c)$ // see table 12
8. $\mathsf{str}_t \leftarrow [\![[t_1, t_2, t_3, t_{4,1}, t_{4,2}]]\!][\![[\hat{t}_1, \ldots, \hat{t}_N]]\!]$
9. $\mathsf{str}_y \leftarrow \mathsf{str}_c [\![\hat{\mathbf{c}}]\!] pk$
10. $c \leftarrow \mathsf{GetNIZKPChallenge}(\texttt{shuffle-challenge} \,|\, \varphi \,|\, \mathsf{str}_t \mathsf{str}_y)$ // see table 14
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 $i = 1, \ldots, N$: $\hat{t}'_i \leftarrow \hat{c}_i^{-c} \times g^{\hat{s}_i} \times \hat{c}_{i-1}^{s'_i}$
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

- $i \in \mathbb{Z}$: 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. $x \leftarrow \mathsf{SHA256}(\mathtt{ggen} \,|\, i) >> 2$ // i in base 10, output as a big-endian number
2. $b \leftarrow \mathsf{uncompress}(x + \varepsilon)$ // ε: smallest non-negative integer such that $b \in E$
3. $h \leftarrow b^8$
4. If $h \in \{1, g\}$, 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 9: Function $\mathsf{GetGenerator}(i)$ (for Ed25519)

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 10: 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
   - append c_i and a comma to S
3. Return S // $S \in \texttt{string}$

Table 11: 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 12: 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 13: Function 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 14: Function 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.