

Belenios specification

Stéphane Gloudu

Version 1.11

Contents

1	Introduction	2
2	Parties	3
3	Processes	3
3.1	Election setup	3
3.1.1	Basic decryption support	3
3.1.2	Threshold decryption support	4
3.2	Vote	4
3.3	Credential recovery	4
3.4	Tally	5
3.5	Audit	5
3.5.1	During the voting phase	5
3.5.2	After the tally	6
4	Messages	6
4.1	Conventions	6
4.2	Basic types	6
4.3	Common structures	6
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	8
4.5.3	Channels	8
4.5.4	Polynomials	9
4.5.5	Vinputs	9
4.5.6	Voutputs	10
4.5.7	Threshold parameters	10
4.6	Trustees	10
4.7	Credentials	11
4.8	Questions	11
4.8.1	Homomorphic questions	11
4.8.2	Non-homomorphic questions	12
4.9	Elections	12
4.10	Encrypted answers	12
4.10.1	Homomorphic answers	13
4.10.2	Non-homomorphic answers	13
4.11	Proofs of interval membership	14
4.12	Proofs of possibly-blank votes	15
4.12.1	Non-blank votes ($m_0 = 0$)	15
4.12.2	Blank votes ($m_0 = 1$)	16

4.12.3 Verifying proofs	16
4.13 Signatures	17
4.14 Ballots	17
4.15 Encrypted tally	17
4.16 Shuffles	18
4.17 Partial decryptions	19
4.18 Election result	19
5 Default group parameters	20
5.1 Homomorphic group	20
5.2 Non-homomorphic group	21
6 Shuffle algorithms	22

1 Introduction

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

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

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

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

2 Parties

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

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

- the election data E
- the list PK of public keys of the trustees
- the list L of public credentials
- the list B of accepted ballots
- the result of the election **result** (once the election is tallied)

3 Processes

3.1 Election setup

1. \mathcal{A} generates a fresh uuid u and sends it to \mathcal{C}
2. \mathcal{C} generates credentials c_1, \dots, c_n and computes $L = \text{shuffle}(\text{public}(c_1), \dots, \text{public}(c_n))$
3. for $j \in [1 \dots n]$, \mathcal{C} sends c_j to \mathcal{V}_j
4. (optional) \mathcal{C} forgets c_1, \dots, c_n
5. \mathcal{C} sends L to \mathcal{A}
6. \mathcal{A} and $\mathcal{T}_1, \dots, \mathcal{T}_m$ run a key establishment protocol (either 3.1.1 or 4.5)
7. \mathcal{A} creates the election E
8. \mathcal{A} loads E and L into \mathcal{S} and starts it
9. \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 4 is optional. It offers a better protection against ballot stuffing in case \mathcal{C} unintentionally leaks private credentials.

3.1.1 Basic decryption support

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

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

$$y = \prod_{z \in [1 \dots m]} \text{public_key}(k_z)$$

3. for $z \in [1 \dots m]$, \mathcal{T}_z checks that k_z appears in the set of public keys PK of the election of uuid u (the id of the election should be publicly known).

3.1.2 Threshold decryption support

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

1. for $z \in [1 \dots m]$,
 - (a) \mathcal{T}_z generates a **cert** γ_z and sends it to \mathcal{A}
 - (b) \mathcal{A} checks γ_z
2. \mathcal{A} assembles $\Gamma = \gamma_1, \dots, \gamma_n$
3. for $z \in [1 \dots m]$,
 - (a) \mathcal{A} sends Γ to \mathcal{T}_z and \mathcal{T}_z checks it
 - (b) \mathcal{T}_z generates a **polynomial** P_z and sends it to \mathcal{A}
 - (c) \mathcal{A} checks P_z
4. for $z \in [1 \dots m]$, \mathcal{A} computes a **vinput** vi_z
5. for $z \in [1 \dots m]$,
 - (a) \mathcal{A} sends Γ to \mathcal{T}_z and \mathcal{T}_z checks it
 - (b) \mathcal{A} sends vi_z to \mathcal{T}_z and \mathcal{T}_z checks it
 - (c) \mathcal{T}_z computes a **voutput** vo_z and sends it to \mathcal{A}
 - (d) \mathcal{A} checks vo_z
6. \mathcal{A} extracts encrypted decryption keys K_1, \dots, K_m and threshold parameters
7. \mathcal{A} computes the election public key y as specified in section 4.5.4.
8. for $z \in [1 \dots m]$, \mathcal{T}_z checks that γ_z appears in the set of public keys PK of the election of **uuid** u (the id of the election should be publicly known).

3.2 Vote

1. \mathcal{V} gets E
2. \mathcal{V} creates a **ballot** b and submits it to \mathcal{S}
3. \mathcal{S} validates b and adds it to B
4. at any time (even after tally), \mathcal{V} may check that b appears in the list of accepted ballots B

3.3 Credential recovery

If \mathcal{C} has forgotten the private credentials of the voter (optional step 4 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 computes the initial `encrypted_tally` Π_0
2. \mathcal{A} extracts the non-homomorphic ciphertexts from the encrypted tally (see section 4.16):

$$\tilde{\Pi}_0 = \text{nh_ciphertexts}(\Pi_0)$$

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{A} 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{A}
 - (c) \mathcal{A} verifies σ_z and extracts $\tilde{\Pi}_z = \text{ciphertexts}(\sigma_z)$
4. \mathcal{A} merges shuffled non-homomorphic ciphertexts with homomorphic ciphertexts, i.e. builds Π such that:

$$\tilde{\Pi}_m = \text{nh_ciphertexts}(\Pi)$$

5. for $z \in [1 \dots m]$ (or, if in threshold mode, a subset of it of size at least $t + 1$),
 - (a) \mathcal{A} sends Π (and K_z if in threshold mode) to \mathcal{T}_z
 - (b) \mathcal{T}_z generates a `partial_decryption` δ_z and sends it to \mathcal{A}
 - (c) \mathcal{A} verifies δ_z
6. \mathcal{A} 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 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 `field(o)` to access the field `field` of `o`.

4.2 Basic types

- `string`: JSON string
- `uuid`: UUID (either as defined in RFC 4122, or a string of Base58 characters¹ 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

$$\text{proof} = \left\{ \begin{array}{l} \text{challenge} : \mathbb{Z}_q \\ \text{response} : \mathbb{Z}_q \end{array} \right\} \quad \text{ciphertext} = \left\{ \begin{array}{l} \text{alpha} : \mathbb{G} \\ \text{beta} : \mathbb{G} \end{array} \right\}$$

¹Base58 characters are: 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmlnopqrstuvwxyz

4.4 Verification keys

$$\begin{aligned} \text{public_key} &= \mathbb{G} & \text{private_key} &= \mathbb{Z}_q \\ \text{trustee_public_key} &= \left\{ \begin{array}{ll} \text{pok} & : \text{proof} \\ \text{public_key} & : \text{public_key} \end{array} \right\} \end{aligned}$$

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. `challenge` = $\mathcal{H}_{\text{pok}}(X, A) \bmod q$
4. `response` = $w + x \times \text{challenge} \bmod q$

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

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

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

1. compute $A = g^{\text{response}} / y^{\text{challenge}}$
2. check that `challenge` = $\mathcal{H}_{\text{pok}}(\text{public_key}, A) \bmod q$

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{A} as a proxy.

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

123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxy

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 message M (as a `string`), computes a signature and produces a `signed_msg`. For the signature, we use a (Schnorr-like) non-interactive zero-knowledge proof.

$$\text{signed_msg} = \left\{ \begin{array}{ll} \text{message} & : \text{string} \\ \text{signature} & : \text{proof} \end{array} \right\}$$

To compute the signature,

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

To verify a signature using a verification key vk ,

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

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.

$$\text{encrypted_msg} = \left\{ \begin{array}{l} \text{alpha} : \mathbb{G} \\ \text{beta} : \mathbb{G} \\ \text{data} : \text{string} \end{array} \right\}$$

To compute the `encrypted_msg`:

1. pick random $r, s \in \mathbb{Z}_q$
2. compute $\text{alpha} = g^r$
3. compute $\text{beta} = \text{ek}^r \times g^s$
4. compute `data` as the hexadecimal encoding of the (symmetric) encryption of M using AES in CCM mode with $\text{SHA256}(\text{key} \parallel g^s)$ as the key and $\text{SHA256}(\text{iv} \parallel g^r)$ 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 $\text{SHA256}(\text{key} \parallel \text{beta} / (\text{alpha}^{\text{dk}}))$
2. compute the initialization vector as $\text{SHA256}(\text{iv} \parallel \text{alpha})$
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.

$$\text{cert} = \text{signed_msg} \quad \text{cert_keys} = \left\{ \begin{array}{l} \text{verification} : \mathbb{G} \\ \text{encryption} : \mathbb{G} \end{array} \right\}$$

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 `send(sk, recipient, message)`, and can be transmitted using a third-party (such as the election administrator).

$$\text{channel_msg} = \left\{ \begin{array}{l} \text{recipient} : \mathbb{G} \\ \text{message} : \text{string} \end{array} \right\}$$

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.

$$\text{polynomial} = \left\{ \begin{array}{l} \text{polynomial} : \text{string} \\ \text{secrets} : \text{string}^* \\ \text{coefexps} : \text{coefexps} \end{array} \right\}$$

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

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

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

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

$$\text{raw_polynomial} = \{ \text{polynomial} : \mathbb{Z}_q^* \}$$

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

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

$$\text{secret} = \{ \text{secret} : \mathbb{Z}_q \}$$

filled with s_{ij}

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

$$\text{coefexps} = \text{signed_msg} \quad \text{raw_coefexps} = \{ \text{coefexps} : \mathbb{G}^* \}$$

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

The public key of the election will be:

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

4.5.5 Vinputs

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

$$\text{vinput} = \left\{ \begin{array}{l} \text{polynomial} : \text{string} \\ \text{secrets} : \text{string}^* \\ \text{coefexps} : \text{coefexps}^* \end{array} \right\}$$

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 $\text{secret}(P_1)_j, \dots, \text{secret}(P_m)_j$
- the **coefexps** field is $\text{coefexps}(P_1), \dots, \text{coefexps}(P_m)$

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 $i = 1, \dots, m$,

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

4.5.6 Voutputs

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

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

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

- `private_key` is set to `send(skj, ekj, Sj)`, where S_j is \mathcal{T}_j 's (private) decryption key:

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

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

The administrator checks vo_j as follows:

- check that:

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

- check `pok(public_key(voj))`

4.5.7 Threshold parameters

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

$$\text{threshold_parameters} = \left\{ \begin{array}{l} \text{threshold} : \mathbb{I} \\ \text{certs} : \text{cert}^* \\ \text{coefexps} : \text{coefexps}^* \\ \text{verification_keys} : \text{trustee_public_key}^* \end{array} \right\}$$

The administrator fills it as follows:

- `threshold` is set to $t + 1$
- `certs` is set to $\Gamma = \gamma_1, \dots, \gamma_m$
- `coefexps` is set to the same value as the `coefexps` field of `vinputs`
- `verification_keys` is set to `public_key(vo1), \dots, public_key(vom)`

4.6 Trustees

$$\text{trustees} = \text{trustee_kind}^*$$

$$\text{trustee_kind} = [\text{"Single"}, \text{trustee_public_key}] \mid [\text{"Pedersen"}, \text{threshold_parameters}]$$

A `trustees` structure is associated to each election. Such a structure is a list of either a single verification key as described in section 4.4, or threshold parameters as described in section 4.5. Each item describes how a partial decryption is computed: either a specific (mandatory) verification key is used to compute a share, or a subset of a set of (optional) verification keys are used to compute a share.

The generality of this definition allows to mix mandatory and optional trustees during decryption. For example, in an election with 3 mandatory trustees, the `trustees` structure will look like:

```
[["Single",...],["Single",...],["Single",...]]
```

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

```
[["Single",...],["Pedersen",...]]
```

4.7 Credentials

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

```
123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

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 $53 - c_1 \bmod 53$.

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

- c as password;
- HMAC-SHA256 (RFC 2104, FIPS PUB 180-2) as pseudorandom function;
- the `uuid` (either interpreted as a 16-byte array in the RFC 4122 case, or directly itself in the Base58 case) of the election as salt;
- 1000 iterations

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

4.8 Questions

$$\text{question_h} = \left\{ \begin{array}{l} \text{answers} : \text{string}^* \\ \text{?blank} : \mathbb{B} \\ \text{min} : \mathbb{I} \\ \text{max} : \mathbb{I} \\ \text{question} : \text{string} \end{array} \right\} \quad \begin{array}{l} \text{question_nh} = \left\{ \begin{array}{l} \text{answers} : \text{string}^* \\ \text{question} : \text{string} \end{array} \right\} \\ \text{question_gen} = \left\{ \begin{array}{l} \text{type} : \text{string} \\ \text{value} : \text{json} \end{array} \right\} \end{array}$$

$$\text{question} = \text{question_h} \mid \text{question_gen}$$

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

$$\text{election} = \left\{ \begin{array}{l} \text{description} : \text{string} \\ \text{name} : \text{string} \\ \text{public_key} : \text{wrapped_pk} \\ \text{questions} : \text{question}^* \\ \text{uuid} : \text{uuid} \end{array} \right\}$$

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

$$\text{embedding} = \left\{ \begin{array}{l} \text{padding} : \mathbb{I} \\ \text{bits_per_int} : \mathbb{I} \end{array} \right\}$$

$$\text{group} = \left\{ \begin{array}{l} g : \mathbb{G} \\ p : \mathbb{N} \\ q : \mathbb{N} \\ \text{?embedding} : \text{embedding} \end{array} \right\} \quad \text{wrapped_pk} = \left\{ \begin{array}{l} \text{group} : \text{group} \\ y : \mathbb{G} \end{array} \right\}$$

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

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

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

4.10 Encrypted answers

$$\text{answer_h} = \left\{ \begin{array}{l} \text{choices} : \text{ciphertext}^* \\ \text{individual_proofs} : \text{iproof}^* \\ \text{overall_proof} : \text{iproof} \\ \text{?blank_proof} : \text{proof}^2 \end{array} \right\}$$

$$\text{answer_nh} = \left\{ \begin{array}{l} \text{choices} : \text{ciphertext} \\ \text{proof} : \text{proof} \end{array} \right\}$$

$$\text{answer} = \text{answer_h} \mid \text{answer_nh}$$

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

4.10.1 Homomorphic answers

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

More concretely, each weight $m \in [0 \dots 1]$ 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 individual proof that $m \in [0 \dots 1]$ is computed by running `iprove(S, r, m, 0, 1)` (see section 4.11).

When a blank vote is not allowed, `overall_proof` proves that $M \in [\min \dots \max]$ and is computed by running `iprove(S, R, M - min, min, \dots, 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 $m_0 = 1 \vee m_\Sigma \in [\min \dots \max]$.

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:

$$\xi = \text{group_encode}_{\kappa, p}([v_1, \dots, v_n]) = (((v_1 \times 2^\kappa + v_2) \times 2^\kappa + \dots) \times 2^\kappa + v_n) \times 2^p + \varepsilon$$

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. `alpha` = g^r
 3. `beta` = $y^r \xi$

where y is the election public key;

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

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

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

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

4.11 Proofs of interval membership

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

Given a pair (α, β) of group elements, one can prove that it has the form $(g^r, y^r g^{M_i})$ with $M_i \in [M_0, \dots, M_k]$ by creating a sequence of proofs π_0, \dots, π_k with the following procedure, parameterised by a group element S :

1. for $j \neq i$:
 - (a) create π_j with a random challenge and a random response
 - (b) compute
$$A_j = \frac{g^{\text{response}}}{\alpha^{\text{challenge}}} \quad \text{and} \quad B_j = \frac{y^{\text{response}}}{(\beta/g^{M_j})^{\text{challenge}}}$$
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) challenge(π_i) = $\mathcal{H}_{\text{iprove}}(S, \alpha, \beta, A_0, B_0, \dots, A_k, B_k) - \sum_{j \neq i} \text{challenge}(\pi_j) \pmod q$
 - (d) response(π_i) = $w + r \times \text{challenge}(\pi_i) \pmod q$

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

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

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 $\text{iprove}(S, r, i, M_0, \dots, M_k)$.

The proof is verified as follows:

1. for $j \in [0 \dots k]$, compute

$$A_j = \frac{g^{\text{response}(\pi_j)}}{\alpha^{\text{challenge}(\pi_j)}} \quad \text{and} \quad B_j = \frac{y^{\text{response}(\pi_j)}}{(\beta/g^{M_j})^{\text{challenge}(\pi_j)}}$$

2. check that

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

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, \dots, m_n and their associated secrets.

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

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

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

6. compute `response`(π_0) = $w - r_0 \times \text{challenge}(\pi_0) \pmod q$

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

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

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 `challenge`(π_0) and `response`(π_0) in \mathbb{Z}_q
2. compute $A_0 = g^{\text{response}(\pi_0)} \times \alpha_0^{\text{challenge}(\pi_0)}$ and $B_0 = y^{\text{response}(\pi_0)} \times (\beta_0/g)^{\text{challenge}(\pi_0)}$
3. for $j > 0$ and $j \neq i$:
 - (a) create π_j with a random `challenge` and a random `response` in \mathbb{Z}_q
 - (b) compute $A_j = g^{\text{response}} \times \alpha_\Sigma^{\text{challenge}}$ and $B_j = y^{\text{response}} \times (\beta_\Sigma/g^{M_j})^{\text{challenge}}$
4. pick a random $w \in \mathbb{Z}_q$
5. compute $A_i = g^w$ and $B_i = y^w$
6. compute

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

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

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

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

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 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 4.12.1, but exchanging subscripts 0 and Σ everywhere except in the call to $\mathcal{H}_{\text{bproof0}}$.

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

1. for $j > 0$:
 - (a) create π_j with a random `challenge` and a random `response` in \mathbb{Z}_q
 - (b) compute $A_j = g^{\text{response}} \times \alpha_\Sigma^{\text{challenge}}$ and $B_j = y^{\text{response}} \times (\beta_\Sigma/g^{M_j})^{\text{challenge}}$
2. pick a random $w \in \mathbb{Z}_q$
3. compute $A_0 = g^w$ and $B_0 = y^w$
4. compute

$$\text{challenge}(\pi_0) = \mathcal{H}_{\text{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) - \sum_{j>0} \text{challenge}(\pi_j) \pmod q$$

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

4.12.3 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 $A_0 = g^{\text{response}(\pi_0)} \times \alpha_0^{\text{challenge}(\pi_0)}$ and $B_0 = y^{\text{response}(\pi_0)} \times \beta_0^{\text{challenge}(\pi_0)}$
2. compute $A_\Sigma = g^{\text{response}(\pi_\Sigma)} \times \alpha_\Sigma^{\text{challenge}(\pi_\Sigma)}$ and $B_\Sigma = y^{\text{response}(\pi_\Sigma)} \times \beta_\Sigma^{\text{challenge}(\pi_\Sigma)}$
3. check that

$$\mathcal{H}_{\text{bproof0}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) = \text{challenge}(\pi_0) + \text{challenge}(\pi_\Sigma) \pmod q$$

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 $A_0 = g^{\text{response}(\pi_0)} \times \alpha_0^{\text{challenge}(\pi_0)}$ and $B_0 = y^{\text{response}(\pi_0)} \times (\beta_0/g)^{\text{challenge}(\pi_0)}$
2. for $j > 0$, compute

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

3. check that

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

4.13 Signatures

$$\text{signature} = \left\{ \begin{array}{l} \text{public_key} : \text{public_key} \\ \text{challenge} : \mathbb{Z}_q \\ \text{response} : \mathbb{Z}_q \end{array} \right\}$$

Each ballot contains a (Schnorr-like) digital signature to avoid ballot stuffing. The signature needs a credential c and uses all the `ciphertexts` $\gamma_1, \dots, \gamma_l$ that appear in the ballot. It is computed as follows:

1. compute $s = \text{secret}(c)$
2. pick a random $w \in \mathbb{Z}_q$
3. compute $A = g^w$
4. `public_key` = g^s
5. `challenge` = $\mathcal{H}_{\text{signature}}(\text{public_key}, A, \gamma_1, \dots, \gamma_l) \bmod q$
6. `response` = $w - s \times \text{challenge} \bmod q$

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

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

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

Signatures are verified as follows:

1. compute $A = g^{\text{response}} \times \text{public_key}^{\text{challenge}}$
2. check that `challenge` = $\mathcal{H}_{\text{signature}}(\text{public_key}, A, \gamma_1, \dots, \gamma_l) \bmod q$

4.14 Ballots

$$\text{ballot} = \left\{ \begin{array}{l} \text{answers} : \text{answer}^* \\ \text{election_hash} : \text{string} \\ \text{election_uuid} : \text{uuid} \\ \text{signature} : \text{signature} \end{array} \right\}$$

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

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

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

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

4.15 Encrypted tally

$$\begin{aligned} \text{ciphertexts_h} &= \text{ciphertext}^* & \text{ciphertexts_nh} &= \text{ciphertext}^* \\ \text{encrypted_tally} &= (\text{ciphertexts_h} | \text{ciphertexts_nh})^* \end{aligned}$$

A so-called *encrypted tally* is constructed out of the accepted ballots B_1, \dots, B_n . It is an array $[C_1, \dots, C_m]$ 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:

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

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:

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

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

$$\text{encrypted_tally} \approx \text{ciphertext}^{**}$$

4.16 Shuffles

If the election has non-homomorphic questions, let us say n out of m ($1 \leq n \leq m$), non-homomorphic ciphertexts must be shuffled. They are first extracted from the encrypted tally a : if i_1, \dots, i_n are the indices of the non-homomorphic questions,

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

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 $b' = \text{nh_ciphertexts}(a')$.

Shuffles are done in the same way as the CHVote system². 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:

$$\text{shuffle} = \left\{ \begin{array}{ll} \text{ciphertexts} & : \text{ciphertext}^{**} \\ \text{proofs} & : \text{shuffle_proof}^* \end{array} \right\}$$

which uses the following auxiliary types:

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

For each non-homomorphic question i :

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

²See version 1.3.2 of the CHVote System Specification at [8]

4.17 Partial decryptions

$$\text{partial_decryption} = \left\{ \begin{array}{l} \text{decryption_factors} : \mathbb{G}^{**} \\ \text{decryption_proofs} : \text{proof}^{**} \end{array} \right\}$$

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*:

$$\text{decryption_factors}_{i,j} = \text{alpha}(a'_{i,j})^x$$

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

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

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

$$\mathcal{H}_{\text{decrypt}}(X, A, B) = \text{SHA256}(\text{decrypt} | X | A, B) \pmod q$$

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

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

1. compute

$$A = \frac{g^{\text{response}}}{\text{public_key}(k)^{\text{challenge}}} \quad \text{and} \quad B = \frac{\text{alpha}(a'_{i,j})^{\text{response}}}{\text{decryption_factors}_{i,j}^{\text{challenge}}}$$

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

4.18 Election result

$$\text{result} = \left\{ \begin{array}{l} \text{num_tallied} : \mathbb{I} \\ \text{encrypted_tally} : \text{encrypted_tally} \\ \text{?shuffles} : \text{shuffle}^* \\ \text{partial_decryptions} : \text{partial_decryption}^* \\ \text{result} : (\mathbb{I}^* | \mathbb{I}^{**})^* \end{array} \right\}$$

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

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

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

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

$$F_{i,j} = \prod_{z \in \mathcal{I}} (\text{partial_decryptions}_{z,i,j})^{\lambda_z^{\mathcal{I}}}$$

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

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

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

- if question i is homomorphic,

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

where j represents an answer. The discrete logarithm can be easily computed because it is bounded by `num_tallied`;

- if question i is non-homomorphic,

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

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 Default group parameters

5.1 Homomorphic group

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

```

p = 20694785691422546
401013643657505008064922989295751104097100884787057374219242
717401922237254497684338129066633138078958404960054389636289
796393038773905722803605973749427671376777618898589872735865
049081167099310535867780980030790491654063777173764198678527
273474476341835600035698305193144284561701911000786737307333
564123971732897913240474578834468260652327974647951137672658
693582180046317922073668860052627186363386088796882120769432
366149491002923444346373222145884100586421050242120365433561
201320481118852408731077014151666200162313177169372189248078
507711827842317498073276598828825169183103125680162072880719

g = 2402352677501852
209227687703532399932712287657378364916510075318787663274146
353219320285676155269678799694668298749389095083896573425601
900601068477164491735474137283104610458681314511781646755400
527402889846139864532661215055797097162016168270312886432456
663834863635782106154918419982534315189740658186868651151358
576410138882215396016043228843603930989333662772848406593138
406010231675095763777982665103606822406635076697764025346253
773085133173495194248967754052573659049492477631475991575198
775177711481490920456600205478127054728238140972518639858334
115700568353695553423781475582491896050296680037745308460627

q = 78571733251071885
079927659812671450121821421258408794611510081919805623223441

```

The additional output of the generation algorithm is:

```

domain_parameter_seed = 478953892617249466
166106476098847626563138168027
716882488732447198349000396592
020632875172724552145560167746
counter = 109

```

5.2 Non-homomorphic group

The group described in the previous section is not suitable for encoding non-homomorphic answers (the `group_encode` function of section 4.10.2). Therefore, we use a different group if the election

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

```

p = 32317006071311007
    300338913926423828248817941241140239112842009751400741706634
    354222619689417363569347117901737909704191754605873209195028
    853758986185622153212175412514901774520270235796078236248884
    246189477587641105928646099411723245426622522193230540919037
    680524235519125679715870117001058055877651038861847280257976
    054903569732561526167081339361799541336476559160368317896729
    073178384589680639671900977202194168647225871031411336429319
    536193471636533209717077448227988588565369208645296636077250
    268955505928362751121174096972998068410554359584866583291642
    136218231078990999448652468262416972035911852507045361090559
g = 2
q = 16158503035655503
    650169456963211914124408970620570119556421004875700370853317
    177111309844708681784673558950868954852095877302936604597514
    426879493092811076606087706257450887260135117898039118124442
    123094738793820552964323049705861622713311261096615270459518
    840262117759562839857935058500529027938825519430923640128988
    027451784866280763083540669680899770668238279580184158948364
    536589192294840319835950488601097084323612935515705668214659
    768096735818266604858538724113994294282684604322648318038625
    134477752964181375560587048486499034205277179792433291645821
    068109115539495499724326234131208486017955926253522680545279

```

Additionally, its embedding field is set to:

$$\left\{ \begin{array}{l} \text{padding} = 8 \\ \text{bits_per_int} = 8 \end{array} \right\}$$

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 [8], and are given here for self-completeness. We also give the `CheckShuffleProof` algorithm, used to check a proof produced by `GenShuffleProof`. For more explanations on these algorithms, please refer to the CHVote System Specification.

Input

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

Algorithm

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

Table 1: Function $\text{GenShuffle}(\mathbf{e}, y)$ **Input**

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

Algorithm

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

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

Input

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

Algorithm

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

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

Input

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

Algorithm

1. $h \leftarrow \text{GetSecondaryGenerator}()$, $\mathbf{h} \leftarrow \text{GetGenerators}(N)$ // see tables 6 and 7
2. $(\mathbf{c}, \mathbf{r}) \leftarrow \text{GenPermutationCommitment}(\psi, \mathbf{h})$ // see table 9
3. $\text{str}_c \leftarrow \llbracket \mathbf{e} \rrbracket \llbracket \mathbf{e}' \rrbracket \llbracket \mathbf{c} \rrbracket$ // see table 10
4. $\mathbf{u} \leftarrow \text{GetNIZKPChallenges}(N, \text{shuffle-challenges} | \text{str}_c)$ // see table 11
5. For $i = 1, \dots, N$: $u'_i \leftarrow u_{j_i}$
6. $\mathbf{u}' \leftarrow [u'_1, \dots, u'_N]$
7. $(\hat{\mathbf{c}}, \hat{\mathbf{r}}) \leftarrow \text{GenCommitmentChain}(h, \mathbf{u}')$ // see table 12
8. For $i = 1, \dots, 4$: pick ω_i at random in \mathbb{Z}_q
9. For $i = 1, \dots, N$: 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 $i = 1, \dots, N$: $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \hat{c}_{i-1}^{\omega'_i}$
14. $t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), [\hat{t}_1, \dots, \hat{t}_N])$, $\text{str}_t \leftarrow \llbracket [t_1, t_2, t_3, t_{4,1}, t_{4,2}] \rrbracket \llbracket [\hat{t}_1, \dots, \hat{t}_N] \rrbracket$
15. $y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, $\text{str}_y \leftarrow \text{str}_c \llbracket \hat{\mathbf{c}} \rrbracket pk$ // pk taken as a number in base 10
16. $c \leftarrow \text{GetNIZKPChallenge}(\text{shuffle-challenge} | \text{str}_t \text{str}_y)$ // see table 13
17. $\bar{r} \leftarrow \sum_{i=1}^N r_i \pmod q$, $s_1 \leftarrow \omega_1 + c \times \bar{r} \pmod q$
18. $v_N \leftarrow 1$
19. For $i = N - 1, \dots, 1$: $v_i \leftarrow u'_{i+1} v_{i+1} \pmod q$
20. $\hat{r} \leftarrow \sum_{i=1}^N \hat{r}_i v_i \pmod q$, $s_2 \leftarrow \omega_2 + c \times \hat{r} \pmod q$
21. $\tilde{r} \leftarrow \sum_{i=1}^N r_i u_i \pmod q$, $s_3 \leftarrow \omega_3 + c \times \tilde{r} \pmod q$
22. $r' \leftarrow \sum_{i=1}^N r'_i u_i \pmod q$, $s_4 \leftarrow \omega_4 + c \times r' \pmod q$
23. For $i = 1, \dots, N$: $\hat{s}_i \leftarrow \hat{\omega}_i + c \times \hat{r}_i \pmod q$, $s'_i \leftarrow \omega'_i + c \times u'_i \pmod q$
24. $s \leftarrow (s_1, s_2, s_3, s_4, [\hat{s}_1, \dots, \hat{s}_N], [s'_1, \dots, s'_N])$
25. $\pi \leftarrow (t, s, \mathbf{c}, \hat{\mathbf{c}})$
26. Return π // $\pi \in \text{shuffle_proof}$

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

Input

- $\pi \in \text{shuffle_proof}$: shuffle proof
- $\mathbf{e} = [e_1, \dots, e_N] \in \text{ciphertext}^N$: encrypted answers to one question; we will denote by α_i and β_i the contents of e_i
- $\mathbf{e}' = [e'_1, \dots, e'_N] \in \text{ciphertext}^N$: 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

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. $h \leftarrow \text{GetSecondaryGenerator}(), \mathbf{h} \leftarrow \text{GetGenerators}(N)$ // see tables 6 and 7
6. $\text{str}_c \leftarrow \llbracket \mathbf{e} \rrbracket \llbracket \mathbf{e}' \rrbracket \llbracket \mathbf{c} \rrbracket$ // see table 10
7. $\mathbf{u} \leftarrow \text{GetNIZKPChallenges}(N, \text{shuffle-challenges} | \text{str}_c)$ // see table 11
8. $\text{str}_t \leftarrow \llbracket [t_1, t_2, t_3, t_{4,1}, t_{4,2}] \rrbracket \llbracket [\hat{t}_1, \dots, \hat{t}_N] \rrbracket$
9. $\text{str}_y \leftarrow \text{str}_c \llbracket \hat{\mathbf{c}} \rrbracket pk$ // pk taken as a number in base 10
10. $c \leftarrow \text{GetNIZKPChallenge}(\text{shuffle-challenge} | \text{str}_t \text{str}_y)$ // see table 13
11. $\bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i$
12. $u \leftarrow \prod_{i=1}^N u_i \pmod q$
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 $i = 1, \dots, N$: $\hat{t}'_i \leftarrow \hat{c}_i^{-c} \times g^{s_i} \times \hat{c}_{i-1}^{s'_i}$
22. Return $(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3) \wedge (t_{4,1} = t'_{4,1}) \wedge (t_{4,2} = t'_{4,2}) \wedge [\bigwedge_{i=1}^N (\hat{t}_i = \hat{t}'_i)]$

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

Algorithm

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

Table 6: Function `GetSecondaryGenerator()`**Input**

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

Algorithm

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

Table 7: Function `GetGenerators(N)`

Input

- $i \in \mathbb{Z}$: number of the independent generator to get

State (shared between all runs)

- $\mathcal{X} \in \mathcal{P}(\mathbb{N} \times \mathbb{G})$ (initialized to \emptyset): generators to avoid

Algorithm

1. $c \leftarrow (p-1)/q$ // typically, $c = 2$
2. $x \leftarrow \text{SHA256}(\text{ggen} \parallel i)$ // 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 $\text{GetGenerator}(i)$ (for a multiplicative subgroup of a finite field)**Input**

- $\psi = [j_1, \dots, j_N] \in \Psi_N$: permutation
- $\mathbf{h} = [h_1, \dots, h_N] \in \mathbb{G}^N$: independent generators

Algorithm

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

Table 9: Function $\text{GenPermutationCommitment}(\psi, \mathbf{h})$

<p>Input</p> <ul style="list-style-type: none"> • $\mathbf{e} = [e_1, \dots, e_N] \in \text{ciphertext}^N$: array of ciphertexts, or • $\mathbf{c} = [c_1, \dots, c_N] \in \mathbb{G}^N$: array of group elements <p>Algorithm</p> <ol style="list-style-type: none"> 1. set S to the empty string 2. For $i = 1, \dots, N$: <ul style="list-style-type: none"> • append $\text{alpha}(e_i)$, a comma, $\text{beta}(e_i)$ and a comma to S, or // in base 10 • append c_i and a comma to S // in base 10 3. Return S // $S \in \text{string}$
--

Table 10: Functions $\llbracket \mathbf{e} \rrbracket$ and $\llbracket \mathbf{c} \rrbracket$

<p>Input</p> <ul style="list-style-type: none"> • $N \in \mathbb{N}$: number of ciphertexts • $S \in \text{string}$: challenge string <p>Algorithm</p> <ol style="list-style-type: none"> 1. $H \leftarrow \text{SHA256}(S)$ // output interpreted as an hexadecimal string 2. For $i = 0, \dots, N - 1$: <ol style="list-style-type: none"> (a) $T \leftarrow \text{SHA256}(i)$ // input taken as decimal, output interpreted as hexadecimal (b) $u_i \leftarrow \text{SHA256}(HT) \bmod q$ // output interpreted as big-endian 3. $\mathbf{u} \leftarrow [u_0, \dots, u_{N-1}]$ 4. Return \mathbf{u} // $\mathbf{u} \in \mathbb{Z}_q^N$

Table 11: Function $\text{GetNIZPKChallenges}(N, S)$

<p>Input</p> <ul style="list-style-type: none"> • $c_0 \in \mathbb{G}$: initial commitment • $\mathbf{u} = [u_1, \dots, u_N] \in \mathbb{Z}_q^N$: public challenges <p>Algorithm</p> <ol style="list-style-type: none"> 1. For $i = 1, \dots, N$: <ol style="list-style-type: none"> (a) Pick r_i at random in \mathbb{Z}_q (b) $c_i \leftarrow g^{r_i} \times c_{i-1}^{u_i}$ 2. $\mathbf{c} \leftarrow [c_1, \dots, c_N]$ 3. $\mathbf{r} \leftarrow [r_1, \dots, r_N]$ 4. Return (\mathbf{c}, \mathbf{r}) <p style="text-align: right;">// $\mathbf{c} \in \mathbb{G}^N, \mathbf{r} \in \mathbb{Z}_q^N$</p>

Table 12: Function $\text{GenCommitmentChain}(c_0, \mathbf{u})$

<p>Input</p> <ul style="list-style-type: none"> • $S \in \text{string}$: challenge string <p>Algorithm</p> <ol style="list-style-type: none"> 1. $c \leftarrow \text{SHA256}(S) \bmod q$ // output interpreted as a big-endian number 2. Return c // $c \in \mathbb{Z}_q$
--

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

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, Wrocław, 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] P. Gaudry. Some ZK security proofs for Belenios. working paper or preprint, 2017.
- [8] 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>.
- [9] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1991*, pages 129–140, 1991.