Bern University of Applied Sciences, CH-2501 Biel, Switzerland

UniVote System Specification

Version 0.5.2

Rolf Haenni

21.05.2013

On behalf of the student unions of the University of Bern (SUB), the University of Zürich (VSUZH), and the Bern University of Applied Sciences (VSBFH).

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.1	14.07.2012	Rolf Haenni	Initial Draft.
0.2	15.08.2012	Rolf Haenni	Complete revision of initial draft.
0.3	10.09.2012	Rolf Haenni	Includes now the case of late registrations.
0.4	13.03.2013	Rolf Haenni	Improved ballot acceptance and closing the urn.
0.5	04.04.2013	Rolf Haenni	Revised section on zero-knowledge proofs.
0.5.1	30.04.2013	Rolf Haenni	Improved notational consistency.
0.5.2	21.05.2013	Rolf Haenni	Two important notational corrections.

Contents

1. Cryptographic Primitives & Encodings

1.1. Cryptographic Primitives

The UniVote system is based on several cryptographic building blocks. Apart from standard ElGamal encryption and decryption, we also need Schnorr signatures, threshold decryptions, non-interactive zero-knowledge proofs of knowledge, verifiable exponentiation and re-encryption mix-nets, an anonymous channel, and an append-only public bulletin board. These building blocks will be briefly described below.

1.1.1. ElGamal Cryptosystem

The *ElGamal cryptosystem* is based on a multiplicative cyclic group $(G_q, \cdot, 1)$ of order q, for which the decisional Diffie-Hellman assumption (DDH) is believed to hold [1]. The most common choice for such a group is the subgroup of quadratic residues $G_q \subset \mathbb{Z}_p^*$ of prime order q, where $p = 2q + 1$ is a *safe prime*. Typically, p is chosen to be large enough (1024–2048 bits) to resist index-calculus and other methods of solving the discrete logarithm problem. The public parameters of an ElGamal cryptosystem are thus p, q, and a generator g of $G_q = \langle g \rangle$. A suitable generator can be found by picking an arbitrary value $\gamma \in \mathbb{Z}_p^*$ and by checking that $g = \gamma^2$ is different from 1.

An ElGamal key pair is a tuple (x, y), where $x \in_R \mathbb{Z}_q$ is the randomly chosen private decryption key and $y = g^x \in G_q$ the corresponding public encryption key. If $m \in G_q$ denotes the plaintext to encrypt, then

$$Enc_y(m, r) = (g^r, m \cdot y^r) \in G_q \times G_q \tag{1.1}$$

is the ElGamal encryption of m with randomization $r \in_R \mathbb{Z}_q$.[1] Note that its bit-length is twice the bit-length of p. For a given encryption $E = (a, b) = Enc_y(m, r)$, m can be recovered by using the private decryption key x to compute

$$Dec_x(E) = a^{-x} \cdot b = m. \tag{1.2}$$

Note that m can also be recovered by $m = b \cdot y^{-r}$ in case the randomization r is known.

The ElGamal encryption function is *homomorphic* with respect to multiplication, which means that the component-wise multiplication of two ciphertexts yields an encryption of the product of respective plaintexts:

$$Enc_y(m_1, r_1) \cdot Enc_y(m_2, r_2) = Enc_y(m_1 \cdot m_2, r_1 + r_2). \tag{1.3}$$

[1] For improved efficiency, we can pick a randomization r with a reduced, but large enough bit-length to resist birthdate attacks on discrete logarithms (160–512 bits). Furthermore, we can pre-compute both parts of an ElGamal encryption prior to knowing the plaintext m.

In a homomorphic cryptosystem like ElGamal, a given encryption $E = Enc_y(m, r)$ can be *re-encrypted* by multiplying E with an encryption of the neutral element 1. The resulting re-encryption,

$$ReEnc_y(E, r') = E \cdot Enc_y(1, r') = Enc_y(m, r + r'), \tag{1.4}$$

is clearly an encryption of m with a fresh randomization $r + r'$.

Practical applications often require the plaintext to be selected from $\mathbb{Z}_q$ rather than G_q. With a safe prime p, we can use the following mapping $G : \mathbb{Z}_q \to G_q$ to encode any integer plaintext $m' \in \mathbb{Z}_q$ by a group element $m \in G_q$, which can then be encrypted as described above:

$$m = G(m') = \begin{cases} m' + 1, & \text{if } (m' + 1)^q = 1, \\ p - (m' + 1), & \text{otherwise.} \end{cases} \tag{1.5}$$

When we obtain $m \in G_q$ from decrypting the ciphertext, we can reconstruct $m' \in \mathbb{Z}_q$ by applying the inverse function $G^{-1} : G_q \to \mathbb{Z}_q$ to m:

$$m' = G^{-1}(m) = \begin{cases} m - 1, & \text{if } m \leq q, \\ (p - m) - 1, & \text{otherwise.} \end{cases} \tag{1.6}$$

Note that by adding such an encoding to the ElGamal cryptosystem, it is no longer homomorphic with respect to plaintexts in $\mathbb{Z}_q$ (but re-encryptions can still be computed in the same way as explained above).

1.1.2. Schnorr Signatures

The *Schnorr signature scheme* has a setting similar to the ElGamal cryptosystem. It is based on a multiplicative cyclic group $(G_q, \cdot, 1)$ of order q, for which the discrete logarithm problem (DLP) is believed to be intractable in the random oracle model [4]. The most common choice is a *Schnorr group*, a subgroup $G_q \subset \mathbb{Z}_p^*$ of prime order q, where $p = kq + 1$ is a prime large enough (1024–2048 bits) to resist methods for solving the discrete logarithm problem, while q is large enough (160–512 bits) to resist birthday attacks on discrete logarithm problems. The public parameters of a Schnorr signature scheme are thus p, q, and a generator g of $G_q = \langle g \rangle$. A suitable generator can be found by picking a random value $\gamma \in_R \mathbb{Z}_p^*$ and by checking that $g = \gamma^k$ is different from 1. Furthermore, all involved parties must agree on a cryptographic hash function $H : \{0, 1\}^* \to \mathbb{Z}_q$.[2]

An Schnorr signature key pair is a tuple (sk, vk), where $sk \in_R \mathbb{Z}_q$ is the randomly chosen private signature key and $vk = g^{sk} \in G_q$ the corresponding public verification key. If $m \in \{0, 1\}^*$ denotes an arbitrary message to sign and $r \in_R \mathbb{Z}_q$ a randomly selected value, then

$$Sign_{sk}(m, r) = (a, r - a \cdot sk) \in \mathbb{Z}_q \times \mathbb{Z}_q, \text{ where } a = H(m || g^r), \tag{1.7}$$

[2] We can choose any cryptographic hash with the desired security properties by applying modulo q to the integer interpretation of its hash value. For example, $H(x) = \text{SHA-256}(x) \bmod q$.

is the Schnorr signature of m. Note that its bit-length is twice the bit-length of q. Using the public verification key vk, a given signature $S = (a, b) = Sign_{sk}(m, r)$ for message m can be verified by computing

$$Verify_{vk}(m, S) = \begin{cases} accept, & \text{if } a = H(m||g^b \cdot vk^a), \\ reject, & \text{otherwise.} \end{cases} \tag{1.8}$$

1.1.3. Digital Certificates

Let $m = id||k||t||CA$ denote a particular type of message, where id is a unique identifier of the holder of a public encryption or signature key k, t a timestamp, and CA the identifier of a *certificate authorithy* with public verification key vk_{CA} and private signature key sk_{CA}. Let $S_{id} = Sign_{sk_{CA}}(id||k||t||CA, r)$ be a digital signature of m for some randomization $r \in_R \mathbb{Z}_q$. Then

$$Z_{id} = Certify_{sk_{CA}}(id, k, t) = (id, k, t, CA, S_{id}) \tag{1.9}$$

is *digital certificate* of k issued by CA at time t. It can be verified by checking that $Verify_{vk_{CA}}(Z_{id}) = Verify_{vk_{CA}}(id||k||t||CA, S_{id}) = accept$.

1.1.4. Zero-Knowledge Proofs of Knowledge

A *zero-knowledge proof* is a cryptographic protocol, where the *prover* P tries to convince the *verifier* V that a mathematical statement is true, but without revealing any information other than the truth of the statement. A *proof of knowledge* is a particular proof allowing P to demonstrate knowledge of a secret information involved in the mathematical statement.

a) Non-Interactive Preimage Proof

One of the most fundamental zero-knowledge proofs of knowledge is the *preimage proof.* Let $(X, +, 0)$ be an additively written and $(Y, \cdot, 1)$ a multiplicatively written group of finite order, and let $\phi : X \to Y$ a one-way group homomorphism. If P knows the preimage $a \in X$ (the *private input*) of a publicly known value $b = \phi(a) \in Y$ (the *public input*), then proving knowledge of a is achieved with the following non-interactive version of the Σ-*protocol.* To generate the proof, P performs the following steps:

1. Choose $\omega \in_R X$ uniformly at random.
2. Compute $t = \phi(\omega)$.
3. Compute $c = H(b, t) \bmod q$, for $q = |\text{image}(\phi)|$.
4. Compute $s = \omega + c \cdot a$.

The pair $\pi = (t, s)$ is the resulting proof which can be published without revealing any information about a. Note that $\text{image}(\phi) = Y$ holds in many concrete instantiations of the preimage proof, and this implies $q = |Y|$. To verify π, the V computes $c = H(b, t) \bmod q$ and checks if $\phi(s) \overset{?}{=} t \cdot b^c$ holds.

b) Examples

Knowledge of Discrete Logarithm (Schnorr)

- Let g be a generator of G_q
- Let $c = g^m$ be a publicly known commitment of $m \in \mathbb{Z}_q$
- P proves knowledge of m using the Σ-protocol for:

$$\begin{aligned} a &= m, \\ b &= c, \\ \phi(x) &= g^x, \end{aligned}$$

where $\phi : \underbrace{\mathbb{Z}_q}_{X} \to \underbrace{G_q}_{Y}$

Equality of Discrete Logarithms

- Let g_1 and g_2 be generators of G_q
- Let $c_1 = g_1^m$ and $c_2 = g_2^m$ be public commitments of $m \in \mathbb{Z}_q$
- P proves knowledge of m using the Σ-protocol for:

$$\begin{aligned} a &= m, \\ b &= (c_1, c_2), \\ \phi(x) &= (g_1^x, g_2^x), \end{aligned}$$

where $\phi : \underbrace{\mathbb{Z}_q}_{X} \to \underbrace{G_q \times G_q}_{Y}$

- Note that $t = (t_1, t_2)$

c) Composition of Preimage Proofs

AND Composition

- Consider n one-way group homomorphism $\phi_i : X_i \to Y_i$
- Let $b_1, \dots, b_n$ be publicly known, where $b_i = \phi_i(a_i)$
- P proves knowledge of $a_1, \dots, a_n$ using the Σ-protocol for:

$$\begin{aligned} a &= (a_1, \dots, a_n), \\ b &= (b_1, \dots, b_n), \\ \phi(x_1, \dots, x_n) &= (\phi_1(x_1), \dots, \phi_n(x_n)), \end{aligned}$$

where $\phi : \underbrace{X_1 \times \cdots \times X_n}_{X} \to \underbrace{Y_1 \times \cdots \times Y_n}_{Y}$

- Note that $\omega = (\omega_1, \ldots, \omega_n)$, $t = (t_1, \ldots, t_n)$, $s = (s_1, \ldots, s_n)$, which implies proofs of size $O(n)$

Equality Proof

- Consider n one-way group homomorphism $\phi_i : X \rightarrow Y_i$
- Let $b_1, \ldots, b_n$ be publicly known, where $b_i = \phi_i(a)$
- P proves knowledge of a using the Σ-protocol for:

$$\begin{aligned} &a, \\ &b = (b_1, \ldots, b_n), \\ &\phi(x) = (\phi_1(x), \ldots, \phi_n(x)), \end{aligned}$$

  where $\phi : X \rightarrow \underbrace{Y_1 \times \cdots \times Y_n}_{Y}$
- Note that $t = (t_1, \ldots, t_n)$, which implies proofs of size $O(n)$

1.1.5. Threshold Cryptosystem

A cryptosystem such as ElGamal is called threshold cryptosystem, if the private decryption key x is shared among n parties, and if the decryption can be performed by a threshold number of parties $t \leq n$ without explicitly reconstructing x and without disclosing any information about the individual key shares x_i. A general threshold version of the ElGamal cryptosystem results from sharing the private key x using Shamir's secret sharing scheme [3, 5]. To avoid the need for a trusted third party to generate the shares of the private key, it is possible to let the n parties execute a distributed key generation protocol [2]. We do not further introduce these techniques here, but we will assume their application throughout this paper, for example by saying that some parties jointly generate a private key or that they jointly decrypt a ciphertext.

A threshold cryptosystem, which is limited to the particular case of $t = n$, is called *distributed cryptosystem*. A simple distributed version of the ElGamal cryptosystem results from setting $x = \sum_i x_i$. To avoid that x gets publicly known, each of the n parties secretly selects its own key share $x_i \in_R \mathbb{Z}_q$ and publishes $y_i = g^{x_i}$ as a commitment of x_i. The product $y = \prod_i y_i = g^{\sum_i x_i} = g^x$ is then the common public encryption key. If $E = (a, b) = Enc_y(m, r)$ is a given encryption, then m can be jointly recovered if each of the n parties computes $a_i = a^{-x_i}$ using its own key share x_i. The resulting product $a^{-x} = \prod_i a_i$ can then be used to derive $m = Dec_x(E) = a^{-x} \cdot b$ from b.[3] Instead of performing this simple operation in parallel, it is also possible to perform essentially the same operation sequentially in form of a *partial decryption function* $Dec'_{x_i}(E) = (a, a^{-x_i} \cdot b)$. Applying Dec'_{x_i} "removes" from E the public key share y_i by transforming it into a new encryption $E' = Dec'_{x_i}(E)$ for a new public key $y \cdot y_i^{-1}$. If all public key shares are removed in this way (in an arbitrary order), we obtain a trivial encryption (a, m) from which m can be extracted.

[3] Alternatively, each party may compute $m_i = Dec_{x_i}(E) = a^{-x_i} \cdot b$ by applying the normal ElGamal decryption function. The plaintext message can then be recovered by $m = b^{1-n} \cdot \prod_i m_i$.

To guarantee the correct outcome of a threshold or distributed decryption, all involved must prove that they followed the protocol properly. In the case of the above distributed version of the ElGamal cryptosystem, each party must deliver two types of non-interactive zero-knowledge proofs:

- $\mathit{NIZKP}\{(x_j) : y_j = g^{x_j}\}$, to prove knowledge of the discrete logarithm of y_j after committing to x_j,
- $\mathit{NIZKP}\{(x_j) : y_j = g^{x_j} \wedge a_j = a^{-x_j}\}$, to prove equality of the discrete logarithms of y_j and a_j^{-1} after computing a_j.

Note that the first proof seems to be subsumed by the second proof, but it is important to provide the first proof along with y_j to guarantee the correctness of y *before* using it as a public encryption key.

If $\{E_1, \ldots, E_N\}$ is a batch of encryptions $E_i = (a_i, b_i)$ to decrypt and $a_{ij} = a_i^{-x_j}$ the corresponding partial decryptions, then it is more efficient to provide a single combined proof,

$$\mathit{NIZKP}\{(x_j) : y_j = g^{x_j} \wedge (\bigwedge_i a_{ij} = a_i^{-x_j})\}, \tag{1.10}$$

instead of N individual proofs of the second type. As discussed in Subsection **??**, a combined proof like this can be implemented efficiently as a batch proof.

1.1.6. Verifiable Mix-Nets

not yet implemented

2. UniBoard

2.1. Overview

UniBoard is an highly configurable implementation of an public bulletin board. For UniVote2 we use a configuration that supports following properties:

- Multipurpose
- Sectioned
- Structured
- Typed
- Access-controlled
- Notifying / Active notification
- Append-only
- Certified reading
- Certified posting
- Searchable

2.2. Properties

2.2.1. UniBoard basic operations

UniBoard supports two basic operations.

- Post$(m, A) : B$
  This operation allows to publish on the *UB* a message $m \in \mathcal{M}$ and additional arguments $A = (\alpha_1, \ldots, \alpha_n)$, where $\alpha_i \in \mathcal{A}_i$. The *UB* might add additional arguments $B = (\beta_1, \ldots, \beta_k)$, where $\beta_j \in \mathcal{B}_j$, and stores the tuple (m, A, B). The *UB* returns the additional arguments B. The arguments in A and B depend on the properties supported by the *UB*.
- Get$(Q) : R$
  This operation allows to get the published data from the *UB* as result R. The result R for a query $Q = (\mathcal{A}'_1, \ldots, \mathcal{A}'_n, \mathcal{B}'_1, \ldots, \mathcal{B}'_k)$, where $\mathcal{A}'_i \subseteq \mathcal{A}_i$ and $\mathcal{B}'_j \subseteq \mathcal{B}_j$, is the set of tuples (m, A, B) such that $\alpha_i \in \mathcal{A}'_i$ and $\beta_j \in \mathcal{B}'_j$. Q enables one to retrieve only the information one is interested in by reducing $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ and $\mathcal{B}'_1, \ldots, \mathcal{B}'_k$ to the desired values. If $\mathcal{A}'_i$ or $\mathcal{B}'_j$ is missing in Q it is assumed to be equal to $\mathcal{A}_i$ or $\mathcal{B}_j$ respectively.

a query

2.2.2. Multipurpose

Multipurpose allows multiple applications to use the same UB. This is achieved by adding the application identifier a_i to A. For Q a_i is mandatory.

2.2.3. Sectioned

Inside a board with this property, an application can be partitioned into multiple sections. E.g.: separate data for different customers into individual sections. For this property, the section identifier e_j is added to A. An application must consist of at least one section. For Q e_j is mandatory.

2.2.4. Structured

In a structure board, a type identifier T_k is added to the list of parameters A when posting a message. For the query Q, T_k is optional.

2.2.5. Typed

The typed property requires the structured property. Moreover, for each type of message, a schema of the message structure must be provided for the board verifies the content structure of every incoming message.

2.2.6. Access-controlled

- Post$(message, t, s)$: -
- Get() : $\mathcal{R}_{UB}$ where $(m, s, t) \in \mathcal{R}_{UB}$

2.2.7. Notifying

- Post$(message, a_i)$: -
- Get(a_i) : $\mathcal{M}_i$

2.2.8. Append-Only

- Post$(message, a_i)$: -
- Get(a_i) : $\mathcal{M}_i$

2.2.9. Certified Reading

- Post($message, a_i$) : -
- Get(a_i) : $\mathcal{M}_i$

2.2.10. Certified Posting

- Post(m) Upon posting your message m the UB creates the signature $S_m(m, t_{UB})$ including your message and a current timestamp t_{UB}. Timestamp and signature are returned to you.
- Get()

2.2.11. Searchable

- Post($message, a_i$) : -
- Get(a_i) : $\mathcal{M}_i$

3. UniCert

4. UniVote

4.1. Overview

Involved parties:

Certificate Authority. CA

Election Administration. EA

Election Coordinator. EC

Uniboard. UB

Talliers. $T_1, \ldots, T_r$

Mixers. $M_1, \ldots, M_m$

Voters. $V_1, \ldots, V_n$

Number of ballots: $N \leq n$

4.2. Detailed Protocol Specification

4.2.1. Public Parameters

The following parameters are assumed to be known in advance and not to change over time.

Schnorr signature scheme:

- $p = 161931481198080639220214033595931441094586304918402813506510547237223787775475425991443924977419330663170224569788019900180050114468430413908687329871251101280878786588515668012772798298511621634145464600626619548823238185390034868354933050128115662663653841842699535282987363300852550784188180264807606304297$ (1024 Bits)
- $q = (p-1)/k = 65133683824381501983523684796057614145070427752690897588060462960319251776021$ (256 Bits)
- $g = 109291242937709414881219423205417309207119127359359243049468707782004862682441897432780127734395596275377218236442035534825283725782836026439537687695084410797228793004739671835061419040912157583607422965551428749149162882960112513332411954585778903685207256083057895070357159920203407651236651002676481874709$ (1024 Bits)

Hash function: (used in Schnorr signatures and zero-knowledge proofs)

- $H(x) = \text{SHA-256}(x) \bmod q$

4.2.2. Public Identifiers and Keys

Certificates for the following identifiers are assumed to be available in a public certificate directory.

a) Registration System

Certificate authority:

- Identifier: CA
- Public certificate: Z_{CA}, self-signed or certified by public certification authority at time t
- Public verification key: vk_{CA}
- Private signature key: sk_{CA}

b) Election System

Election Coordinator:

- Identifier: EC
- Public certificate: Z_{EC}, signed by CA at time t
- Public verification key: vk_{EC}
- Private signature key: sk_{EC}

Bulletin Board:

- Identifier: UB
- Public certificate: Z_{UB}, signed by CA at time t
- Public verification key: vk_{UB}
- Private signature key: sk_{UB}

c) Election Trustees

Talliers: (for $1 \leq j \leq r$)

- Identifier: T_j
- Public certificate: Z_j, signed by CA at time t_j
- Public verification key: vk_j
- Private signature key: sk_j

Mixers: (for $1 \leq k \leq m$)

- Identifier: M_k
- Public certificate: Z_k, signed by CA at time t_k
- Public verification key: vk_k
- Private signature key: sk_k

d) Election Participants

Election administration:

- Identifier: EA
- Public certificate: Z_{EA}, signed by CA at time t
- Public verification key: vk_{EA}
- Private signature key: sk_{EA}

Voters: (for $1 \leq i \leq n$)

- Identifier: V_i
- Personal credentials: $cred_i$ issued by the CA or an affiliated identity provider.

4.2.3. At all times

UB notifies registered listeners about published messages. Anybody can register itself as a listener. (This section might be changed to a description of the properties of the bulletin board(certified read,multi tenant))

4.2.4. Registration

Registration can take place at any time, possibly long before an election starts. A registered voter can use the private signature key multiple times.

a) First-Time Registration

V_i performs the following steps:

1. Choose $sk_i \in_R \mathbb{Z}_q$ uniformly at random.
2. Compute $vk_i = g^{sk_i} \bmod p$.
3. Generate $\pi_{sk_i} = \mathit{NIZKP}\{(sk_i) : vk_i = g^{sk_i} \bmod p\}$ to prove knowledge of sk_i (see Subsection 4.4.1 for details).
4. Send $(V_i, cred_i, vk_i, \pi_{sk_i})$ to CA.

Upon receipt, CA performs the following steps:

5. Check validity of $(V_i, cred_i)$.
6. Check correctness of π_{sk_i} (see Subsection 4.4.1 for details).
7. Determine current timestamp t_i.
8. Compute H(V_i)
9. Compute $Z_i = \mathit{Certify}_{sk_{CA}}(H(V_i), vk_i, t_i) = (H(V_i), vk_i, t_i, CA, S_i)$.
10. Publish Z_i on UB.

b) Registration Renewal

The above procedure allows V_i to renew the registration at any time, simply by performing the same steps again. The new certificate $\bar{Z}_i = (V_i, \bar{vk}_i, \bar{t}_i, CA, \bar{S}_i)$ will contain a timestamp $\bar{t}_i > t_i$, which will implicitly disqualify any former certificate $Z_i = (V_i, vk_i, t_i, CA, S_i)$ in current or future elections. CA should warn V_i before issuing a new certificate.

4.2.5. Election Setup

The following tasks can be performed in advance, possibly long before the election starts.

a) Initialization

Upon request from EA to run an election, EC performs the following steps:

1. Choose unique *election identifier id*.
2. Select Z_{EA} from the UB. Check that $\mathit{Verify}_{vk_{CA}}(Z_{EA}) = \mathit{accept}$.
3. Generate signature $S_{EA} = \mathit{Sign}_{sk_{EC}}(id||Z_{EA})$.
4. Publish (EC, id, Z_{EA}, S_{EA}) on UB.

b) Election Definition

EA performs the following steps:

1. Define textual description of the election event $descr$.
2. Define security parameter ℓ (e.g. $\ell = 2048$ bits)
3. Define talliers $T = \{T_1 \ldots, T_r\}$.
4. For each $T_j \in T$, select Z_j from UB and check that $Verify_{vk_{CA}}(Z_j) = accept$. Let $\mathcal{Z}_T = \{Z_j : 1 \leq j \leq r\}$.
5. Define mixers $M = \{M_1 \ldots, M_m\}$.
6. For each $M_k \in M$, select Z_k from UB and check that $Verify_{vk_{CA}}(Z_k) = accept$. Let $\mathcal{Z}_M = \{Z_k : 1 \leq k \leq m\}$.
7. Generate signature $S_{descr} = Sign_{sk_{EA}}(id||descr||\ell||\mathcal{Z}_T||\mathcal{Z}_M)$.
8. Publish $(EA, id, descr, \ell, \mathcal{Z}_T, \mathcal{Z}_M, S_{descr})$ on UB.

c) Parameter Generation

EC performs the following steps:[1]

1. Define ElGamal parameters P (of length ℓ bits), $Q = (P-1)/2$, and $G \in \{2, 3, 4\}$ (the smallest possible value). We use capital letters to distinguish them from Schnorr parameters.
2. Generate signature $S_{PQG} = Sign_{sk_{EC}}(id||P||Q||G)$.
3. Publish $(EC, id, P, Q, G, S_{PQG})$ on UB.

d) Distributed Key Generation

Each $T_j \in T$ performs the following steps:

1. Check that $Verify_{vk_{EC}}(id||P||Q||G, S_{PQG}) = accept$.
2. Choose $x_j \in_R \mathbb{Z}_Q$ uniformly at random.
3. Compute $y_j = G^{x_j} \bmod P$.
4. Generate $\pi_{x_j} = NIZKP\{(x_j) : y_j = G^{x_j} \bmod P\}$ to prove knowledge of x_j (see Subsection 4.4.3 for details).
5. Generate signature $S_{y_j} = Sign_{sk_j}(id||y_j||\pi_{x_j})$.
6. Publish $(T_j, id, y_j, \pi_{x_j}, S_{y_j})$ on UB.

EC performs the following steps:

7. For each $T_j \in T$, do the following:

[1] The same set of parameters may be used for several elections with the same security parameter.

    a) Check that $\mathit{Verify}_{vk_j}(id||y_j||\pi_{x_j}, S_{y_j}) = \mathit{accept}$.

    b) Check correctness of π_{x_j} (see Subsection 4.4.3 for details).

8. Compute $y = \prod_j y_j \bmod P$.
9. Generate signature $S_y = \mathit{Sign}_{sk_{EC}}(id||y)$.
10. Publish (EC, id, y, S_y) on UB.

e) Constructing the Election Generator

Let $g_0 = g$ the publicly known generator of the Schnorr signature scheme. Each $M_k \in M$ performs the following steps (in ascending order for $1 \leq k \leq m$):

1. Choose $\alpha_k \in_R \mathbb{Z}_q$ at random.
2. Compute blinded generator $g_k = g_{k-1}^{\alpha_k} \bmod p$.
3. Generate $\pi_{\alpha_k} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \bmod p\}$ to prove knowledge of α_k (see Subsection 4.4.4 for details).
4. Generate signature $S_{g_k} = \mathit{Sign}_{sk_k}(id||g_k||\pi_{\alpha_k})$.
5. Publish $(M_k, id, g_k, \pi_{\alpha_k}, S_{g_k})$ on UB.

EC performs the following steps:

6. For each $M_k \in M$, do the following:

    a) Check that $\mathit{Verify}_{vk_k}(id||g_k||\pi_{\alpha_k}, S_{g_k}) = \mathit{accept}$.

    b) Check correctness of π_{α_k} (see Subsection 4.4.4 for details).

7. Let $\hat{g} = g_m$ be the *election generator*.
8. Generate signature $S_{\hat{g}} = \mathit{Sign}_{sk_{EC}}(id||\hat{g})$.
9. Publish $(EC, id, \hat{g}, S_{\hat{g}})$ on UB.

4.2.6. Election Preparation

The following tasks are performed shortly before starting the election.

a) Definition of Election Options

EA performs the following steps:

1. Define the set of choices C and a rule set R describing the set $\mathcal{V}^* = \mathit{Votes}(C, R)$ of valid *election options*, where *Votes* is a publicly known function (see Section 4.5.1 for details).
2. Generate signature $S_C = \mathit{Sign}_{sk_{EA}}(id||C||R)$.
3. Publish (EA, id, C, R, S_C) on UB.

b) Publication of Election Data

EC performs the following steps:

1. Check that $\mathit{Verify}_{vk_{EA}}(id||C||R, S_C) = \mathit{accept}$.
2. Generate signature $S_{data} = \mathit{Sign}_{sk_{EC}}(id||EA||descr||P||Q||G||y||\hat{g}||C||R)$.
3. Publish $(EC, id, EA, descr, P, Q, G, y, \hat{g}, C, R, S_{data})$ on UB.

c) Electoral Roll Preparation

EA performs the following steps:

1. Define the *electoral roll* as the set of eligible Voters $V = \{V_1, \ldots, V_n\}$.
2. Compute $H_V = \{H(V_1), \ldots, H(V_n)\}$.
3. Generate signature $S_V = \mathit{Sign}_{sk_{EA}}(id||H_V)$.
4. Publish (EA, id, H_V, S_V) on UB.

EC performs the following steps:

5. Check that $\mathit{Verify}_{vk_{EA}}(id||H_V, S_V) = \mathit{accept}$.
6. For every $H(V_i) \in H_V$, select the most recent Z_i from the public certificate directory. For each Z_i, check that $\mathit{Verify}_{vk_{CA}}(Z_i) = \mathit{accept}$. If yes, add it to the list of registered voters $\mathcal{Z}_V = \{Z_1, \ldots, Z_n\}$.[2]
7. Generate signature $S_V = \mathit{Sign}_{sk_{EC}}(id, \mathcal{Z}_V)$.
8. Publish $(EC, id, \mathcal{Z}_V, S_V)$ on UB.

d) Mixing the Public Verification Keys

Let $VK_0 = \{vk_1, \ldots, vk_n\}$ be the (ordered) set of public verification keys in $\mathcal{Z}_V$. Repeat the following steps for each $M_k \in M$ (in ascending order for $1 \leq k \leq m$):

1. Shuffle the set of public verification keys VK_{k-1} into VK_k:
   a) Compute blinded verification key $vk'_i = vk_i^{\alpha_k}$ for every $vk_i \in VK_{k-1}$.
   b) Choose permutation $\psi_k : [1, n] \rightarrow [1, n]$ uniformly at random.
   c) Let $VK_k = \{vk'_{\psi_k(i)} : 1 \leq i \leq n\} = \mathit{Shuffle}_{\psi_k}(VK_{k-1}, \alpha_k)$ be the new (ordered) set of public verification keys shuffled according to ψ_k.
2. Generate $\pi_{\psi_k} = \mathit{NIZKP}\{(\psi_k, \alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge VK_k = \mathit{Shuffle}_{\psi_k}(VK_{k-1}, \alpha_k)\}$ using Wikström's proof of a shuffle (see Section 4.4.5 for details).
3. Generate signature $S_{VK_k} = \mathit{Sign}_{sk_k}(id||VK_k||\pi_{\psi_k})$.

[2] In many contexts, the number of registered voters is likely to be much smaller than the number of eligible voters.

4. Publish $(M_k, id, VK_k, \pi_{\psi_k}, S_{VK_k})$ on UB.

EC performs the following steps:

5. For each $M_k \in M$, do the following:
   a) Check that $Verify_{vk_k}(id||VK_k||\pi_{\psi_k}, S_{VK_k}) = accept$.
   b) Check correctness of π_{ψ_k} (see Section 4.4.5 for details).
6. Let $VK' = VK_m = \{vk'_{\psi(i)} : 1 \leq i \leq n\}$ for $\psi = \psi_m \circ \cdots \circ \psi_1$.
7. Generate signature $S_{VK'} = Sign_{sk_{EC}}(id, VK')$.
8. Publish $(EC, id, VK', S_{VK'})$ on UB.

4.2.7. Election Period

a) Vote Creation and Casting

Consider the case of $V_i \in V$ creating and casting a vote. To do so, V_i performs the following steps:

1. Retrieve $(EC, id, EA, descr, P, Q, G, y, \hat{g}, C, R, S_{data})$ from UB.[3]
2. Check that $Verify_{vk_{EC}}(id||EA||descr||P||Q||G||y||\hat{g}||C||R, S_{data}) = accept$.
3. Determine the set $\mathcal{V}^* = Votes(C, R)$ of election options.[4]
4. Choose vote $v_i \in \mathcal{V}^*$.
5. Represent v_i as an integer $m'_i = Encode_{C,R}(v_i) \in \mathbb{Z}_Q$.
6. Compute $m_i = G(m'_i) \in G_Q$.
7. Choose $r_i \in_R \mathbb{Z}_Q$ uniformly at random.
8. Compute $E_i = Enc_y(m_i, r_i) = (a_i, b_i)$.
9. Generate $\pi_{r_i} = NIZKP\{(m_i, r_i) : E_i = Enc_y(m_i, r_i)\}$ to prove knowledge of (m_i, r_i). Note that if $E_i = (a_i, b_i) = (G^{r_i}, m_i \cdot y^{r_i})$ is an ElGamal encryption, then this proof is equivalent to the proof $NIZKP\{(r_i) : a_i = G^{r_i}\}$, which implies knowledge of m_i (see Subsection 4.4.6 for details).
10. Generate signature $S_i = Sign_{sk_i}(id||E_i||\pi_{r_i})$ using $\hat{g}$.
11. Compute anonymous verification key $vk'_j = \hat{g}^{sk_i}$, where $j = \psi(i)$.
12. Publish ballot $B_i = (vk'_j, id, E_i, \pi_{r_i}, S_i)$ on UB.

Upon receipt of B_i, UB performs the following tests:

13. Check that vk'_j belongs to an eligible voter: $vk'_j \in VK' \cup \bar{VK}'$.

[3] EA and $descr$ are not explicitly required in the following steps. But it is important for V_i to learn the identity of the election administration and the content of the election.

[4] The set $\mathcal{V}^*$ is not constructed explicitly, it is rather provided implicitly by a proper voting GUI on the voter's client computer.

14. Check that V_i has not previously submitted another ballot:[5]
    a) Check that no ballot on UB contains vk'_j.
    b) If $vk'_j \in \bar{VK}' \cup \hat{VK}'$, check that no other ballot on UB contains a different key of V_i from $\bar{VK}' \cup \hat{VK}'$.
15. Check validity of B_i
    a) Check that $Verify_{vk'_j}(id||E_i||\pi_{r_i}, S_i) = accept$ using $\hat{g}$.
    b) Optional: Check correctness of π_{r_i} (see Subsection 4.4.6 for details).

UB publishes B_i, if all tests succeed.

b) Closing the Electronic Urn

When the election period is over, EC performs the following steps:

1. For each $B_i = (vk'_j, id, E_i, \pi_{r_i}, S_i)$, do the following:
    a) Check that $vk'_j \in VK' \cup \bar{VK}'$.
    b) Check that no other (more recent) ballot contains vk'_j
    c) If $vk'_j \in \bar{VK}' \cup \hat{VK}'$, check that no other (more recent) ballot on UB contains a different key of V_i from $\bar{VK}' \cup \hat{VK}'$.
    d) Check that $Verify_{vk'_j}(id||E_i||\pi_{r_i}, S_i) = accept$ using $\hat{g}$.
    e) Check correctness of π_{r_i} (see Subsection 4.4.6 for details).
2. Let $\mathcal{B}$ be the set of ballot B_i, for which all above checks succeed.
3. Generate signature $S_{\mathcal{B}} = Sign_{sk_{EC}}(id||\mathcal{B})$.
4. Publish $(EC, id, \mathcal{B}, S_{\mathcal{E}})$ on UB.

4.2.8. Mixing and Tallying

a) Mixing the Encryptions

Let $\mathcal{E}_0 = \{E_1, \ldots, E_N\}$, $N \leq n$, be the (ordered) set of encrypted votes in $\mathcal{B}$. Repeat the following steps for each $M_k \in M$ (in ascending order for $1 \leq k \leq m$):

1. Shuffle the set encrypted votes $\mathcal{E}_{k-1}$ into $\mathcal{E}_k$:
    a) Choose $\bar{r}_k = (r_{1k}, \ldots, r_{Nk}) \in_R \mathbb{Z}_q^N$ uniformly at random and compute $E'_i = ReEnc_y(E_i, r_{ik})$ for every $E_i \in \mathcal{E}_{k-1}$.
    b) Choose permutation $\tau_k : [1, N] \to [1, N]$ uniformly at random.

[5] Since re-voting is not supported, only the first ballot counts.

   c) Let $\mathcal{E}_k = \{E'_{\tau_k(i)} : 1 \leq i \leq N\} = \mathit{Shuffle}_{\tau_k}(\mathcal{E}_{k-1}, \bar{r}_k)$ be the new (ordered) set of encrypted votes shuffled according to τ_k.

2. Generate $\pi_{\tau_k} = \mathit{NIZKP}\{(\tau_k, \bar{r}_k) : \mathcal{E}_k = \mathit{Shuffle}_{\tau_k}(\mathcal{E}_{k-1}, \bar{r}_k)\}$ using Wikström's proof of a shuffle (see Section 4.4.7 for details).

3. Generate signature $S_{\mathcal{E}_k} = \mathit{Sign}_{sk_k}(id||\mathcal{E}_k||\pi_{\tau_k})$.

4. Publish $(M_k, id, \mathcal{E}_k, \pi_{\tau_k}, S_{\mathcal{E}_k})$ on UB.

EC performs the following steps:

5. For each $M_k \in M$, do the following:

   a) Check that $\mathit{Verify}_{vk_k}(id||\mathcal{E}_k||\pi_{\tau_k}, S_{\mathcal{E}_k}) = \mathit{accept}$

   b) Check correctness of π_{τ_k} (see Section 4.4.7 for details).

6. Let $\mathcal{E}' = \mathcal{E}_m = \{E'_{\tau(i)} : 1 \leq i \leq N\}$ for $\tau = \tau \circ \cdots \circ \tau_1$.

7. Generate signature $S_{\mathcal{E}'} = \mathit{Sign}_{sk_{EA}}(id||\mathcal{E}')$.

8. Publish $(EC, id, \mathcal{E}', S_{\mathcal{E}'})$ on UB.

b) Decrypting the Votes

Each $T_j \in T$ performs the following steps:

1. Check that $\mathit{Verify}_{vk_{EC}}(id||\mathcal{E}', S_{\mathcal{E}'}) = \mathit{accept}$.

2. Let $\bar{a} = (a_1, \ldots, a_N)$ for $(a_i, b_i) \in \mathcal{E}'$.

3. Compute $\bar{a}_j = (a_{1j}, \ldots, a_{Nj})$, where $a_{ij} = a_i^{-x_j} \bmod P$.

4. Generate $\pi'_{x_j} = \mathit{NIZKP}\{(x_j) : [y_j = G^{x_j} \bmod P] \wedge \left[\bigwedge_i a_{ij} = a_i^{-x_j} \bmod P\right]\}$ to prove knowledge of x_j (see Subsection 4.4.8 for details).

5. Generate signature $S_{\bar{a}_j} = \mathit{Sign}_{sk_j}(id||\bar{a}_j||\pi'_{x_j})$.

6. Publish $(T_j, id, \bar{a}_j, \pi'_{x_j}, S_{\bar{a}_j})$ on UB.

EC performs the following steps:

7. For each $T_j \in T$, do the following:

   a) Check that $\mathit{Verify}_{vk_j}(id||\bar{a}_j||\pi'_{x_j}, S_{\bar{a}_j}) = \mathit{accept}$

   b) Check correctness of π'_{x_j} (see Subsection 4.4.8 for details).

8. For all $1 \leq i \leq N$, do the following:

   a) Compute $m_i = b_i \cdot \prod_j a_{ij} \bmod P$.

   b) Compute $m'_i = G^{-1}(m_i)$.

   c) Compute $v_i = \mathit{Decode}_{C,R}(m'_i)$.

9. Let $\mathcal{V} = \{v_1, \ldots, v_N\} \cap \mathcal{V}^*$ be the list of valid plaintext votes.

10. Generate signature $S_{\mathcal{V}} = Sign_{sk_{EC}}(id||\mathcal{V})$.
11. Publish $(EC, id, \mathcal{V}, S_{\mathcal{V}})$ on UB.

4.3. Late Registration

In principle, the previously described protocol requires the voter to register prior to an election. In some contexts, however, it will be impossible to enforce that *all* voters have registered when the election starts. Those without a registration would then be excluded from casting a vote.

Let V_i be an eligible voter in a current election. This can be tested by checking if $H(V_i) \in H_V$, where V is denotes the set of eligible voters and H_V the corresponding set of hash values as published on UB (see Subsection 4.2.6). A late registration invokes the procedure described below.

4.3.1. Election Period

a) New Registration

Upon notification of a newly registered voter V_i during the election period of the election id, EC performs the following steps:

1. Check if $H(V_i) \in H_V$.
2. Select the new certificate $\bar{Z}_i = (V_i, \bar{vk}_i, \bar{t}_i, CA, \bar{C}_i)$ from the public certificate directory. Check that $Verify_{vk_{CA}}(\bar{Z}_i) = accept$.
3. Generate signature $S_{\bar{Z}_i} = Sign_{sk_{EC}}(id, \bar{Z}_i)$.
4. Publish $(EC, id, \bar{Z}_i, S_{\bar{Z}_i})$ on UB. Let $\bar{\mathcal{Z}}_V$ denote the current set of certificates added during the election period.

Let $\bar{vk}_{i,0} = \bar{vk}_i$ be the new verification key from $\bar{Z}_i$. Repeat the following steps for each $M_k \in M$ (in ascending order for $1 \leq k \leq m$):[6]

5. Compute $\bar{vk}_{i,k} = \bar{vk}_{i,k-1}^{\alpha_k}$.
6. Generate $\pi_{\bar{vk}_{i,k}} = NIZKP\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge \bar{vk}_{i,k} = \bar{vk}_{i,k-1}^{\alpha_k}\}$ (see Subsection 4.4.2 for details).
7. Generate signature $S_{\bar{vk}_{i,k}} = Sign_{sk_k}(id||\bar{vk}_{i,k}||\pi_{\bar{vk}_{i,k}})$.
8. Publish $(M_k, id, \bar{vk}_{i,k}, \pi_{\bar{vk}_{i,k}}, S_{\bar{vk}_{i,k}})$ on UB.

EC performs the following steps:

9. For each $M_k \in M$, do the following:

[6] Note that this procedure corresponds to the borderline case of the general mixing procedure for a single innput public key (with a simplified proof).

a) Check that $\mathit{Verify}_{vk_k}(id||\bar{vk}_{i,k}||\pi_{\bar{vk}_{i,k}}, S_{\bar{vk}_k}) = \mathit{accept}$.

b) Check correctness of $\pi_{\bar{vk}_{i,k}}$ (see Subsection 4.4.2 for details).

10. Let $\bar{vk}'_i = \bar{vk}_{i,m} = \bar{vk}_i^{\alpha}$.
11. Generate signature $S_{\bar{vk}'_i} = \mathit{Sign}_{sk_{EC}}(id,\, V_i,\, \bar{vk}'_i)$.
12. Publish $(EC, id,\, V_i, \bar{vk}'_i, S_{\bar{vk}'_i})$ on UB. Let $\bar{VK}'$ denote the set of all public keys $\bar{vk}'_i$ added during the election period.

b) Late Renewal of Registration

Essentially the same steps are repeated, if the current set $\mathcal{Z}_V \cup \bar{\mathcal{Z}}_V$ contains an earlier certificate $\hat{Z}_i = (V_i, \hat{vk}_i, \hat{t}_i, CA, \hat{C}_i)$ of V_i. Note that the first part of the above procedure is not repeated, since $\hat{Z}_i \in \mathcal{Z}_V \cup \bar{\mathcal{Z}}_V$ implies that $\hat{Z}_i$ has already been verified. Let $\hat{vk}_{i,0} = \hat{vk}_i$ be the former verification key from $\hat{Z}_i$. Repeat the following steps for each $M_k \in M$ (in ascending order for $1 \leq k \leq m$):[7]

1. Compute $\hat{vk}_{i,k} = \hat{vk}_{i,k-1}^{\alpha_k}$.
2. Generate $\pi_{\hat{vk}_{i,k}} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge \hat{vk}_{i,k} = \hat{vk}_{i,k-1}^{\alpha_k}\}$ (see Subsection 4.4.2 for details).
3. Generate signature $S_{\hat{vk}_{i,k}} = \mathit{Sign}_{sk_k}(id||\hat{vk}_{i,k}||\pi_{\hat{vk}_{i,k}})$.
4. Publish $(M_k, id, \hat{vk}_{i,k}, \pi_{\hat{vk}_{i,k}}, S_{\hat{vk}_{i,k}})$ on UB.

EC performs the following steps:

5. For each $M_k \in M$, do the following:

   a) Check that $\mathit{Verify}_{vk_k}(id||\hat{vk}_{i,k}||\pi_{\hat{vk}_{i,k}}, S_{\hat{vk}_{i,k}}) = \mathit{accept}$.

   b) Check correctness of $\pi_{\hat{vk}_{i,k}}$ (see Subsection 4.4.2 for details).

6. Let $\hat{vk}'_i = \hat{vk}_m = \hat{vk}_i^{\alpha}$.
7. Generate signature $S_{\hat{vk}'_i} = \mathit{Sign}_{sk_{EC}}(id,\, V_i,\, \hat{vk}'_i)$.
8. Publish $(EC, id,\, V_i, \hat{vk}'_i, S_{\hat{vk}'_i})$ on UB. Let $\hat{VK}'$ denote the set of all public keys $\hat{vk}'_i$, which have been replaced by a new one during the election period.

[7] Again, this procedure corresponds to the borderline case of the general mixing procedure for a single innput public key (with a simplified proof).

4.4. Details of Proofs

4.4.1. Registration

This is a standard proof of knowledge of discrete logarithm (Schnorr):

$$\pi_{sk_i} = \mathit{NIZKP}\{(sk_i) : vk_i = g^{sk_i} \bmod p\} = (t, c, s).$$

a) Generation

Prover: V_i

1. Choose $\omega \in_R \mathbb{Z}_q$ uniformly at random.
2. Compute $t = g^{\omega_j} \bmod p$.
3. Compute $c = H(vk_i || t || V_i) \bmod q$.
4. Compute $s = \omega + c \cdot sk_i \bmod q$.

b) Verification

1. Check that $c = H(vk_i || t || V_i) \bmod q$.
2. Compute $v = g^s \bmod p$.
3. Compute $w = t \cdot vk_i^c \bmod p$.
4. Check that $v = w$.

4.4.2. Registration Renewal

This is a standard proof of equality of discrete logarithm (Pedersen):

$$\pi_{\bar{vk}_{i,k}} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge \bar{vk}_{i,k} = \bar{vk}_{i,k-1}^{\alpha_k}\} = (\bar{t}, s).$$

a) Generation

Prover: M_k

1. Choose $\omega \in_R \mathbb{Z}_q$ uniformly at random.
2. Compute $\bar{t} = (t_1, t_2) = (g_{k-1}^{\omega} \bmod p, \bar{vk}_{i,k-1}^{\omega} \bmod p)$.
3. Compute $c = H(g_k || \bar{vk}_{i,k} || \bar{t} || M_k) \bmod q$.
4. Compute $s = \omega + c \cdot \alpha_k \bmod q$.

b) Verification

1. Check that $c = H(g_k||\bar{vk}_{i,k}||\bar{t}||M_k) \bmod q$.
2. Compute $\bar{v} = (g_{k-1}^s \bmod p, \bar{vk}_{i,k-1}^s \bmod p)$.
3. Compute $\bar{w} = (t_1 \cdot g_k^c \bmod p, t_2 \cdot \bar{vk}_{i,k}^c \bmod p)$.
4. Check that $\bar{v} = \bar{w}$.

4.4.3. Distributed Key Generation

This is a standard proof of knowledge of discrete logarithm (Schnorr):

$$\pi_{x_j} = \mathit{NIZKP}\{(x_j) : y_j = G^{x_j} \bmod P\} = (t, c, s).$$

a) Generation

Prover: T_j

1. Choose $\omega \in_R \mathbb{Z}_Q$ uniformly at random.
2. Compute $t = G^\omega \bmod P$.
3. Compute $c = H(y_j||t||T_j) \bmod Q$.
4. Compute $s = \omega + c \cdot x_j \bmod Q$.

b) Verification

1. Check that $c = H(y_j||t||T_j) \bmod Q$.
2. Compute $v = G^s \bmod P$.
3. Compute $w = t \cdot y_j^c \bmod P$.
4. Check that $v = w$.

4.4.4. Constructing the Election Generator

This is a standard proof of knowledge of discrete logarithm (Schnorr):

$$\pi_{\alpha_k} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \bmod p\} = (t, c, s).$$

a) Generation

Prover: M_k

1. Choose $\omega \in_R \mathbb{Z}_q$ uniformly at random.
2. Compute $t = g_{k-1}^{\omega} \bmod p$.
3. Compute $c = H(g_k||t||M_k) \bmod q$.
4. Compute $s = \omega + c \cdot \alpha_k \bmod q$.

b) Verification

1. Check that $c = H(g_k||t||M_k) \bmod q$.
2. Compute $v = g_{k-1}^{s} \bmod p$.
3. Compute $w = t \cdot g_k^c \bmod p$.
4. Check that $v = w$.

4.4.5. Mixing the Public Verification Keys

This proof is not yet implemented:

$$\pi_{\psi_k} = \mathit{NIZKP}\{(\psi_k, \alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge VK_k = \mathit{Shuffle}_{\psi_k}(VK_{k-1}, \alpha_k)\}.$$

4.4.6. Vote Creation and Casting

This is a standard proof of knowledge of discrete logarithm (Schnorr):

$$\pi_{r_i} = \mathit{NIZKP}\{(r_i) : a_i = G^{r_i}\} = (t, c, s).$$

a) Generation

Prover: V_i (acting with vk'_j as anonymous identifier)

1. Choose $\omega \in_R \mathbb{Z}_Q$ uniformly at random.
2. Compute $t = G^{\omega} \bmod P$.
3. Compute $c = H(a_i||t||vk'_j) \bmod Q$.
4. Compute $s = \omega + c \cdot r_i \bmod Q$.

b) Verification

1. Check that $c = H(a_i || t || vk'_j) \bmod Q$.
2. Compute $v = G^s \bmod P$
3. Compute $w = t \cdot a_i^c \bmod P$.
4. Check that $v = w$.

4.4.7. Mixing the Encryptions

This proof is not yet implemented:

$$\pi_{\tau_k} = \mathit{NIZKP}\{(\tau_k, \bar{r}_k) : \mathcal{E}_k = \mathit{Shuffle}_{\tau_k}(\mathcal{E}_{k-1}, \bar{r}_k)\}.$$

4.4.8. Decrypting the Votes

This is a general equality proof with multiple functions:

$$\pi'_{x_j} = \mathit{NIZKP}\{(x_j) : [y_j = G^{x_j} \bmod P] \wedge \left[\bigwedge_i a_{ij} = a_i^{-x_j} \bmod P\right]\} = (\bar{t}, c, s).$$

a) Generation

Prover: T_j

1. Choose $\omega \in_R \mathbb{Z}_Q$ uniformly at random.
2. Compute $\bar{t} = (t_0, t_1, \ldots, t_N) = (G^\omega \bmod P, a_1^{-\omega} \bmod P, \ldots, a_N^{-\omega} \bmod P)$.
3. Compute $c = H(y_j || \bar{a}_j || \bar{t} || T_j) \bmod Q$.
4. Compute $s = \omega + c \cdot x_j \bmod Q$.

b) Verification

1. Check that $c = H(y_j || \bar{a}_j || \bar{t} || T_j) \bmod Q$.
2. Compute $\bar{v} = (G^s \bmod P, a_1^{-s} \bmod P, \ldots, a_N^{-s} \bmod P)$.
3. Compute $\bar{w} = (t_0 \cdot y_j^c \bmod P, t_1 \cdot a_{1j}^c \bmod P, \ldots, t_N \cdot a_{Nj}^c \bmod P)$.
4. Check that $\bar{v} = \bar{w}$.

4.5. Encoding Choices, Rules, and Vote

4.5.1. Choices and Rules

To allow a variety of different election types, we consider two finite sets, a set C of possible *election choices* and a set R of *election rules.* For each election choice $c \in C$ in a given election, the election system outputs the number of votes that c has received from the voters. Each election rule in R defines some constraints on how voters can distribute their votes among the election choices. We use $v(c)$ to denote this number for a particular voter. We distinguish three types of election rules:

- *Summation-Rule*: The sum of votes for election choices in a subset $C' \subseteq C$ is within a certain range $[a, b]$, i.e., $\sum_{c \in C'} v(c) \in [a, b]$. Such rules will be denoted by $\Sigma : C' \rightarrow [a, b]$.
- *Forall-Rule*: For each election choice in a subset $C' \subseteq C$, the number of votes is within a certain range $[a, b]$, i.e., $v(c) \in [a, b]$ for all $c \in C'$. Such rules will be denoted by $\forall : C' \rightarrow [a, b]$.
- *Distinctness-Rule*: For each election choice in a subset $C' \subseteq C$, the number of votes is either equal to 0 or unique within C', i.e., $v(c) > 0$ implies $v(c) \neq v(c')$ for all other election choices $c' \in C' \setminus \{c\}$. Such rules will be denoted by $\neq : C'$.

Two or several sets of candidates and sets of rules can be combined to describe multiple elections that run in parallel. We call this operation *composition of elections* and denote it by

$$(C_1, R_1) \circ (C_2, R_2) = (C_1 \cup C_2, R_1 \cup R_2)$$

for two sets of choices C_1, C_2 and corresponding sets of rules R_1, R_2. Note that this can be used to describe party-list elections (see example below).

a) Examples

- Referendum: 1-out-of-2

$$C = \{yes, no\},\ R = \left\{ \begin{array}{c} \Sigma : \{yes, no\} \rightarrow [1, 1] \\ \forall : \{yes, no\} \rightarrow [0, 1] \end{array} \right\}$$

- Referendum with Null Votes:

$$C = \{yes, no\},\ R = \left\{ \begin{array}{c} \Sigma : \{yes, no\} \rightarrow [0, 1] \\ \forall : \{yes, no\} \rightarrow [0, 1] \end{array} \right\}$$

or

$$C = \{yes, no, null\},\ R = \left\{ \begin{array}{c} \Sigma : \{yes, no, null\} \rightarrow [1, 1] \\ \forall : \{yes, no, null\} \rightarrow [0, 1] \end{array} \right\}$$

- Multiple-Choice Referendum / Plurality Voting: 1-out-of-n

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \to [1,1] \\ \forall : \{c_1, \ldots, c_n\} \to [0,1] \end{array} \right\}$$

  Null votes can be handled as shown above.

- Approval Voting: n-out-of-n

$$C = \{c_1, \ldots, c_n\},\ R = \big\{ \forall : \{c_1, \ldots, c_n\} \to [0,1] \big\}$$

- Range Voting: Up to s votes per choice

$$C = \{c_1, \ldots, c_n\},\ R = \big\{ \forall : \{c_1, \ldots, c_n\} \to [0,s] \big\}$$

- Plurality-at-Large Voting / Limited Voting: k-out-of-n

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \to [0,k] \\ \forall : \{c_1, \ldots, c_n\} \to [0,1] \end{array} \right\}$$

- Cumulative Voting: k votes in total, up to $s \leq k$ votes per choice

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \to [0,k] \\ \forall : \{c_1, \ldots, c_n\} \to [0,s] \end{array} \right\}$$

  Note that $k > n$ is allowed here.

- Preferential Voting (Borda Count): Ranks from 1 to n

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{l} \ \forall : \{c_1, \ldots, c_n\} \to [1,n] \\ \neq : \{c_1, \ldots, c_n\} \end{array} \right\}$$

- Preferential Voting (Borda Count): Ranks from 1 to k only

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{l} \ \forall : \{c_1, \ldots, c_n\} \to [1,k] \\ \neq : \{c_1, \ldots, c_n\} \end{array} \right\}$$

- Party-List Election with Cumulation: Composition of cumulative voting over a set of candidates and plurality voting over a set of party-lists

$$\begin{aligned} &C = \{c_1, \ldots, c_n\} \cup \{\ell_1, \ldots, \ell_m\}, \\ &R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \to [0,k] \\ \forall : \{c_1, \ldots, c_n\} \to [0,s] \end{array} \right\} \cup \left\{ \begin{array}{c} \Sigma : \{\ell_1, \ldots, \ell_m\} \to [1,1] \\ \forall : \{\ell_1, \ldots, \ell_m\} \to [0,1] \end{array} \right\} \end{aligned}$$

  Null votes (with respect to party-lists) can be handled as shown above.

4.5.2. Encoding Votes

Let $C = \{c_1, \ldots, c_n\}$ be a set of election choices and $v_i = v(c_i) \leq \rho_i$ the number of votes attributed to c_i by a given voter. With ρ_i we denote the maximum number of votes that the election rules in R allow for c_i. The tuple $v = (v_1, \ldots, v_n) \in \mathcal{V}^*$ represents a valid vote, where $\mathcal{V}^* = \mathit{Votes}(C, R)$ denotes the set of all valid votes for given sets C and R.

To encode $v \in \mathcal{V}^*$ as an integer, consider a bit string of length $B = \sum_{i=1}^{n} b_i$, where $b_i = \lfloor \log_2 \rho_i \rfloor + 1$ denotes the number of bits we reserve for each value v_i. Furthermore, let $B_i = \sum_{j=1}^{i-1} b_j$ be the number of bits in the bit sting *prior* to v_i, i.e., $B_1 = 0$, $B_2 = b_1$, $B_3 = b_1 + b_2$, $\ldots$, $B_{n+1} = B$. The encoding function $\mathit{Encode}_{C,R} : \mathcal{V}^* \rightarrow \{0, \ldots, 2^B{-}1\}$ can then be defined as follows:

$$\mathit{Encode}_{C,R}(v) = \sum_{i=1}^{n} v_i \cdot 2^{B_i}.$$

Since some integers in $\{0, \ldots, 2^B{-}1\}$ do not represent valid votes according to the election rules R, this encoding is not optimal in terms of memory space consumption. In the vast majority of cases, however, be believe that the size of the message space G_Q of the ElGamal cryptosystem is large enough to support this encoding.

To decode an integer representation $x = \mathit{Encode}_{C,R}(v)$ back to $v = (v_1, \ldots, v_n)$, we must decompose the bit string into its components. Mathematically, this decomposition can be written as follows:

$$\mathit{Decode}_{C,R}(x) = (v_1, \ldots, v_n), \text{ where } v_i = \lfloor x/2^{B_i} \rfloor \bmod 2^{b_i}.$$

Bibliography

[1] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *CRYPTO'84, Advances in Cryptology*, LNCS 196, pages 10–18, Santa Barbara, USA, 1984. Springer.

[2] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In J. Stern, editor, *EUROCRYPT'99, 18th International Conference on the Theory and Application of Cryptographic Techniques*, LNCS 1592, pages 295–310, Prague, Czech Republic, 1999.

[3] T. P. Pedersen. A threshold cryptosystem without a trusted party. In D. W. Davies, editor, *EUROCRYPT'91, 10th Workshop on the Theory and Application of Cryptographic Techniques*, volume 547 of *LNCS 547*, pages 522–526, Brigthon, U.K., 1991.

[4] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[5] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.