Assembly Voting X

Stefan Patachi - September 2019

Contents

1 Introduction

This document presents all the technical details of the design of our election system, Assembly Voting X. It describes all the cryptographic mechanisms that are used and all the protocols that the actors of the voting process have to follow.

In section 2 we describe the cryptographic algorithms that we use in our system and the mathematical principles that they rely on. In section 3 we present the entire process that takes part during an election. We define all the protocols that happen between various entities that participate in the election process and also, we specify what cryptographic algorithms each protocol involves. Moreover, we present the protocols in chronological order of how they take place in the election process.

In section 4 we describe how a full auditing process can be performed, what algorithms are included and who are the actors responsible for each step. In the last section 5 we describe what security properties our election system achieves and under what assumptions.

2 Theoretical Background

2.1 Mathematics

2.1.1 Group

In mathematics, a group $\mathcal{G} = (\mathbb{G}, \circ, \text{inv}, e)$ is an algebraic structure consisting of a set $\mathbb{G}$ of elements, a binary operation indicated by symbol $\circ$, a unary operation called **inv** and a neutral element $e \in \mathbb{G}$. The following properties must be satisfied by $\mathcal{G}$:

closure	$x \circ y \in \mathbb{G}$
associativity	$x \circ (y \circ z) = (x \circ y) \circ z$
identity element	$x \circ e = e \circ x = x$
inverse element	$x \circ \mathbf{inv}(x) = e$

for all $x, y, z \in \mathbb{G}$.

If $\mathcal{G}$ has a fifth property called *commutativity* ($x \circ y = y \circ x$), then $\mathcal{G}$ is an *abelian group*.

Moreover, if $\mathcal{G}$ is a *finite group*, then $\mathbb{G}$ has a finite number of elements and we denote $q = |\mathbb{G}|$ as the order of the group. For example, a finite group would be $(\mathbb{Z}_q, +, -, 0)$, where $\mathbb{Z}_q = \{0, 1, ..., q-1\}$, the binary operation is addition modulo q, the inverse operation is negation and the identity element is 0.

The binary operation can be applied on the same element, namely $x \circ x = [2]x$. We define $[k]x$ as the operation $\circ$ applied k times on the element x.

A finite group $\mathcal{G} = (\mathbb{G}, \circ, \text{inv}, e)$ of order q is called *cyclic group*, if there is a group element $g \in \mathbb{G}$, such that $\mathbb{G} = (g, [2]g, [3]g, ..., [q]g)$. In this case, the element g is called the *generator* of $\mathcal{G}$.

2.1.2 Finite Field

A field $\mathcal{F} = (\mathbb{F}, +, \cdot)$ consists of a set $\mathbb{F}$ which is an abelian group in respect to both operations: addition and multiplication. The following properties hold:

- $x + y \in \mathbb{F}$ and $x \cdot y \in \mathbb{F}$
- $(\mathbb{F}, +, -, 0)$ is an abelian group
- $(\mathbb{F}^*, \cdot, ^{-1}, 1)$ is an abelian group
- multiplication is distributive over addition: $x \cdot (y + z) = x \cdot y + x \cdot z$

for all $x, y, z \in \mathbb{F}$.

A finite field is a filed with a finite number of elements, for example the set of integers modulo p, denoted $\mathbb{F}_p$, where p is a prime number.

2.1.3 Elliptic Curve over a Prime Field

We define the elliptic curve E over the prime field $\mathbb{F}_p$ as the set of points

$$E(\mathbb{F}_p) = \{(x, y) \in (\mathbb{F}_p)^2 \mid y^2 = x^3 + ax + b \pmod p\} \cup \{\mathcal{O}\}$$

where a tuple (x, y) represent the coordinates of a point, $\mathcal{O} = (0, 0)$ is the point at infinity and $a, b \in \mathbb{F}_p$.

The elliptic curve $E(\mathbb{F}_p)$ follows a group structure with the following rules:

- $\mathcal{O}$ is the identity element, thus $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E(\mathbb{F}_p)$.
- The inverse operations is point negation, noted $-$. For all $P = (x_P, y_P) \in E(\mathbb{F}_p)$, we define $-P = (x_P, -y_P)$ such that $P + (-P) = \mathcal{O}$.
- The binary operation is point addition, noted $+$. Let $P, Q \in E(\mathbb{F}_p)$. The line through P and Q intersects the elliptic curve in a third point $R = (x_R, y_R) \in E(\mathbb{F}_p)$. The point addition is defined as $P + Q = -R$. The coordinates of R can be computed in the following way:

  $$\begin{aligned} x_R &= \lambda^2 - x_P - x_Q \pmod p \\ y_R &= y_P + \lambda(x_R - x_P) \pmod p \end{aligned}$$

  where λ is the steep of line PQ. The steep can be computed in the following way:

  $$\lambda = \begin{cases} (y_P - y_Q)(x_P - x_Q)^{-1} \pmod p & \text{, if } P \neq Q \\ (3x_P^2 + a)(2y_P)^{-1} \pmod p & \text{, if } P = Q \end{cases}$$

We define the total number of point on the $E(\mathbb{F}_p)$ as N and it can be calculated using *Schoof's algorithm* [1]. Any subgroup of $E(\mathbb{F}_p)$ has an order q which is a divisor of N. In such a case, we define the *cofactor* of the subgroup as $h = \frac{N}{q}$. To find any generator of the subgroup we follow:

- Choose a random point $P \in E(\mathbb{F}_p)$.
- Compute $G = [h]P$.
- If $G = \mathcal{O}$, repeat the process. Otherwise, G is a generator.

In conclusion, we can define our cryptographic cyclic subgroup as:

$$\mathcal{G} = \{P \in E(\mathbb{F}_p) \mid P = [k]G, k \in \mathbb{Z}_q\}$$

where G is the generator and q is the order of the subgroup. We call the integer k a *scalar*.

2.1.4 Elliptic Curve Discrete Logarithm Problem

The *Elliptic Curve Discrete Logarithm Problem* is defined in [2] the following way: Given the elliptic curve domain parameters (p, a, b, G, q, h) and a point $P \in \mathcal{G}$, find the scalar $k \in \mathbb{Z}_p$ such that $P = [k]G$. For an elliptic curve to be cryptographically strong, the ECDLP has to be *computationally infeasible*.

For our cryptosystem we will use a predefined elliptic curve called *secp256k1*, also known as the *Bitcoin Curve* with the following domain parameters:

p	`ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff fffffffe fffffc2f`
a	`0`
b	`7`
$G.x$	`79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798`
$G.y$	`483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8`
q	`ffffffff ffffffff ffffffff fffffffe baaedce6 af48a03b bfd25e8c d0364141`
h	`1`

The equation of the elliptic curve is $y^2 = x^3 + 7$. The cofactor of the curve is 1, that means any point on the curve is part of our cyclic subgroup as well. We assume the Elliptic Curve Discrete Logarithm Problem to be hard in respect to the *secp256k1*. We will reffer to this as the *discrete logarithm assumption*.

2.1.5 Elliptic Curve Point Encoding

Each point on the curve is represented by its x and y coordinate. As presented in section 2.1.3, the y coordinate can be calculated based on the x coordinate, but there would be two possible values for it. Thus, one extra bit of information is required specifying which of the two values is to be used.

An elliptic curve point can be represented as byte array in two ways: *compressed form* or *uncompressed form*. The compressed form contains the byte representation of only the x coordinate to which is prepended a byte, 02 or 03 depending on which value to choose for the y coordinate. The uncompressed form contains the byte representation of both x and y coordinates concatenated together, to which is prepended the byte 04.

In our system, when an elliptic curve point has to be stored in the database, or when it needs to be transferred over the network, or when it is used as input to a hash function, it is represented as byte array in compressed form.

2.1.6 Mapping a message on the Elliptic Curve

An important use case of a cryptographic system is to be able to interpret an arbitrary message (a plain text, a number, an id or even a more complex construction). In elliptic curve context, that means mapping a message into an elliptic curve point in a deterministic way. Additionally, this curve point must be able to be interpreted back as the original message.

We use the algorithm $M \leftarrow \textbf{String2Point}(m)$ (algorithm 1), presented in [2], that can map a plain text $m \in \mathbb{C}^*$ to an elliptic curve point $M \in E(\mathbb{F}_p)$,

where $\mathbb{C}$ is the set of all possible one byte UTF-8 text characters. The byte representation of message m is converted into a large integer m_{BN}, called *BigNum*, and used in the calculation of the x coordinate of the point M, while the y coordinate is computed using the equation of the elliptic curve $y \leftarrow \mathbf{RecoverY}(\mathbf{x})$. Note that the elliptic curve equation might spawn no valid values for y or two values for y. In case y is not valid, it means x is not a valid coordinate to generate a point, otherwise, the algorithm randomly chooses one of the two value for y and continues.

The x coordinate is computed by multiplying m_{BN} with 256, in order to use the right most byte as an *adjusting byte* in case the x coordinate does not correspond to a valid point on the curve, i.e. we keep incrementing x until we find a valid y. To ensure that x is less than the prime of the curve p we introduce the condition that m can be at most 30 characters in length.

By having one *byte space* to find a valid point on the curve, we know from [2] that the probability that all 256 x coordinates generate invalid points is $1/2^{256}$, which we consider acceptable. Formally, the algorithm **String2Point** converts an arbitrary text of maximum 30 characters in length into a valid point on the elliptic curve with negligible failure rate.

Algorithm 1: String2Point(m)

Data: The message $m = (c_1, ..., c_\ell) \in \mathbb{C}^*$
if $\ell > 30$ **then**
  return *failure*
end
$\boldsymbol{b} = (b_1, ..., b_\ell) \leftarrow$ **ByteRepresentationOf**(m)
$m_{\text{BN}} \leftarrow$ **Bytes2BigNum**$(\boldsymbol{b})$
$x_0 \leftarrow m_{\text{BN}} \cdot 256$
for $i \leftarrow 0$ **to** 255 **by** 1 **do**
  $x \leftarrow x_0 + i$
  $y \leftarrow$ **RecoverY**(x) // $y^2 = x^3 + 7$
  $M \leftarrow (x, y)$
  if M *is valid* **then**
    return M // $M \in E(\mathbb{F}_p)$
  end
end
return *failure*

Recovering the text message m from an elliptic curve point M can be done by calling $m \leftarrow$ **Point2String**(M) (algorithm 2). The algorithm tries to interpret as text the byte representation of the x coordinate, disregarding the *adjusting byte* (the right most byte). In case this fails, the algorithm returns a failure state, meaning that point M does not encode a text message.

Having these two algorithms, mapping a message on the Elliptic Curve is a sound procedure as

$m = \textbf{Point2String}(\textbf{String2Point}(m))$, for all $m \in \mathbb{C}^*$, with $|m| \leq 30$.

Algorithm 2: Point2String(M)

Data: The point $M = (x, y) \in E(\mathbb{F}_p) = \mathbb{Z}_p \times \mathbb{Z}_p$
$m_{\text{BN}} \leftarrow x/256$ `// disregard the adjusting byte`
$\boldsymbol{b} = (b_1, ..., b_\ell) \leftarrow \textbf{ByteRepresentationOf}(m_{\text{BN}})$
for $i \leftarrow 1$ **to** ℓ **by** 1 **do**
  $c_i \leftarrow \textbf{Byte2Char}(b_i)$
end
if *all* $c_i \in \mathbb{C}$ **then**
  $m = (c_1, ..., c_\ell)$
else
  return *failure*
end
return m `//` $m \in \mathbb{C}^*$

2.2 Zero Knowledge Proofs

A *zero knowledge proof* (ZKP) is an algorithm by which one party (the *prover*) can prove to another party (the *verifier*) that she knows a secret value x, without disclosing any information about x. A ZKP can be *interactive*, where the prover and the verifier have to collaborate in a protocol for the verifier to get convinced of the proof. A ZKP can also be *non-interactive*. In this case, the prover alone generates a proof that is publicly verifiable, thus convincing any public verifier of its statement.

There exists two algorithms **Prove** for generating a proof and **Verify** for verifying weather a proof validates. A classic proof has a structure of a triple (commitment, challenge and response). In an interactive zero knowledge protocol, a prover commits to a value, the verifier independently and randomly generates a challenge, the prover computes a response based on the challenge received and the verifier checks that the proof validates. The proof can be turned into a non-interactive one, using the *Fiat-Shamir heuristic* as described in [3]. The prover computes alone the challenge, in a deterministic manner, based on the commitment, using a hash function.

2.2.1 Discrete Logarithm Proof

A simple kind of ZKP is the *discrete logarithm proof* that proves knowledge of value x, such that $Y = [x]G$, formally

$$PK[(x) : Y = [x]G]$$

The most intuitive application of this could be to prove the possession of the private key associated with a public key. In order to prove such a fact, a prover and a verifier have to interact in the protocol described in figure 1. The resulting proof would be $PK = (K, c, r)$.

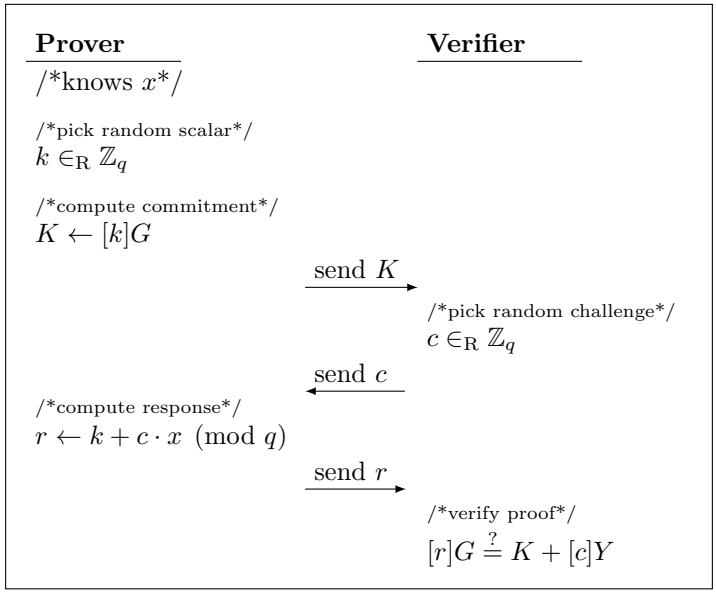

Figure 1: Protocol for proving the discrete logarithm

To generate a *non-interactive proof*, the protocol from figure 2 is turned into two distinct algorithms: one for generating a proof $PK \leftarrow \mathbf{Prove}_G(x)$ (algorithm 3) and another one for validating whether a proof validates $b \leftarrow \mathbf{Verify}_G(PK, Y)$ (algorithm 4) that outputs a bit $b \in \mathbb{B}$ which represents *true* or $false$. The prover generates a commitment, then computes the challenge of the proof, based on that commitment, using a hash function, and finally, computes the response of the proof. The prover publishes the proof, which is universally verifiable. Any public verifier can check whether the proof validates.

Algorithm 3: $\mathbf{Prove}_G(x)$

Data: The base generator $G \in E(\mathbb{F}_p)$
The private key $x \in \mathbb{Z}_q$
$k \in_{\mathrm{R}} \mathbb{Z}_q$
$K \leftarrow [k]G$
$Y \leftarrow [x]G$
$c \leftarrow \mathcal{H}(G||K||Y)$ `// compute challenge by hash function`
$r \leftarrow k + c \cdot x \pmod q$
$PK \leftarrow (K, c, r)$
return PK // $PK \in E(\mathbb{F}_p) \times \mathbb{Z}_q \times \mathbb{Z}_q$

Algorithm 4: Verify$_G(PK, Y)$

Data: The base generator $G \in E(\mathbb{F}_p)$
The proof $PK = (K, c, r) \in E(\mathbb{F}_p) \times \mathbb{Z}_q \times \mathbb{Z}_q$
The public key $Y \in E(\mathbb{F}_p)$
if $c = \mathcal{H}(G||K||Y)$ **and** $[r]G = K + [c]Y$ **then**
  $b \leftarrow 1$ // proof is valid
else
  $b \leftarrow 0$ // proof is invalid
end
return b // $b \in \mathbb{B}$

2.2.2 Discrete Logarithm Equality Proof

A bit more complex ZKP is the *discrete logarithm equality proof* that proves that two different elliptic curve points Y, $P \in E(\mathbb{F}_p)$ have the same elliptic curve discrete logarithm $x \in \mathbb{Z}_q$ in regards to two distinct generators $G, H \in E(\mathbb{F}_p)$.

$$PK[(x) : Y = [x]G \wedge P = [x]H]$$

The proof has the same structure as in previous section $PK = (\boldsymbol{K}, c, r)$, but here the commitment $\boldsymbol{K} = (K_{\mathrm{G}}, K_{\mathrm{H}})$ is a tuple of points, one for each generator that the proof is based on. The updated protocol for proving the discrete logarithm equality is described in figure 2.

Again, to generate the discrete logarithm equality proof in a *non-interactive* manner, we turn the protocol into two algorithms: one for generating the proof $PK \leftarrow$ **ProveEquality**$_{G,H}(x)$ (algorithm 5) and another one for validating a proof $b \leftarrow$ **VerifyEquality**$_{G,H}(PK, Y, P)$ (algorithm 6), where $b \in \mathbb{B}$ represents *true* or *false*. In this case, proof PK is universally verifiable and can be audited by any public verifier.

Algorithm 5: ProveEquality$_{G,H}(x)$

Data: The first base generator $G \in E(\mathbb{F}_p)$
The second base generator $H \in E(\mathbb{F}_p)$
The private key $x \in \mathbb{Z}_q$
$k \in_{\mathrm{R}} \mathbb{Z}_q$
$K_{\mathrm{G}} \leftarrow [k]G$
$K_{\mathrm{H}} \leftarrow [k]H$
$\boldsymbol{K} \leftarrow (K_{\mathrm{G}}, K_{\mathrm{H}})$
$c \leftarrow \mathcal{H}(G||H||K_{\mathrm{G}}||K_{\mathrm{H}}||Y||P)$ // $Y = [x]G$, $P = [x]H$
$r \leftarrow k + c \cdot x \pmod q$
$PK \leftarrow (\boldsymbol{K}, c, r)$
return PK // $PK \in E(\mathbb{F}_p)^2 \times \mathbb{Z}_q \times \mathbb{Z}_q$

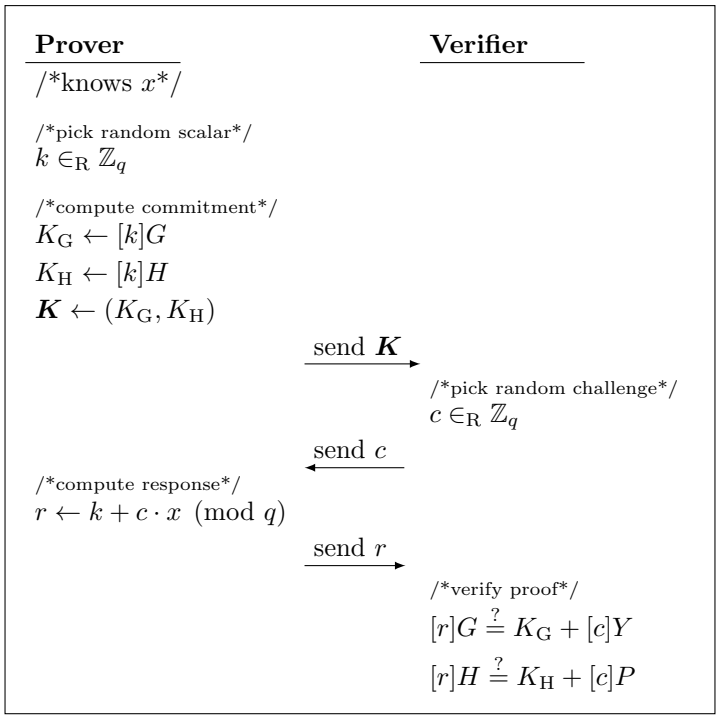

Figure 2: Protocol for proving the discrete logarithm equality

Algorithm 6: VerifyEquality$_{G,H}(PK, Y, P)$

Data: The first base generator $G \in E(\mathbb{F}_p)$
The second base generator $H \in E(\mathbb{F}_p)$
The proof $PK = (\boldsymbol{K}, c, r) \in E(\mathbb{F}_p)^2 \times \mathbb{Z}_q \times \mathbb{Z}_q$
The first public key $Y \in E(\mathbb{F}_p)$
The second public key $P \in E(\mathbb{F}_p)$

if $c = \mathcal{H}(G||H||K_{\mathrm{G}}||K_{\mathrm{H}}||Y||P)$
and $[r]G = K_{\mathrm{G}} + [c]Y$
and $[r]H = K_{\mathrm{H}} + [c]P$ **then**
  $b \leftarrow 1$ `// proof is valid`
else
  $b \leftarrow 0$ `// proof is invalid`
end
return b `//` $b \in \mathbb{B}$

2.2.3 Proof for Multiple Discrete Logarithms

An optimization in proving the discrete logarithm equality between multiple points in regards to their generators has been described in [4]. Using the optimized algorithm for proving that

$$PK[(x) : \bigwedge_{i=0}^{n} Y_i = [x]G_i]$$

we generate the proof $PK = (K, c, r)$ by following the protocol described in figure 3. The optimization consists in the fact that the commitment K is just one point instead of a list of points, regardless of the value of n.

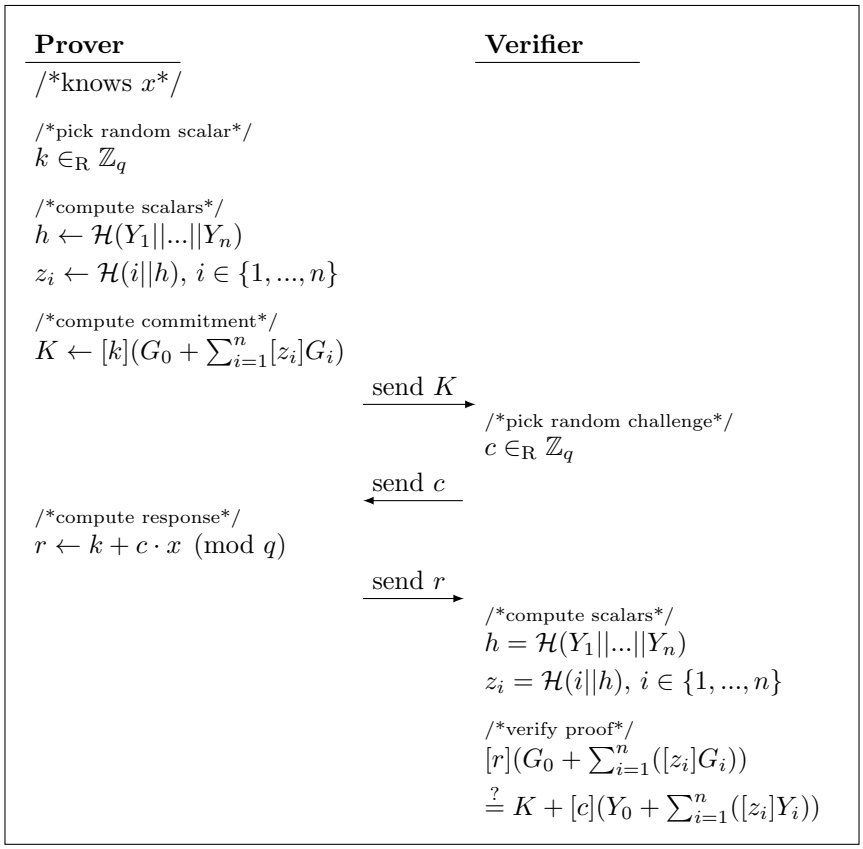

Figure 3: Protocol for proving multiple discrete logarithms

The same as in the discrete logarithm equality case, the proof of multiple discrete logarithms can be turned into an non-interactive one by computing the challenge of the proof based on the commitment, using a hash function. The proof is generated by the algorithm $PK \leftarrow \mathbf{ProveMultiple}_{\boldsymbol{G}}(x)$ (algorithm 7), where $\boldsymbol{G} = (G_0, ...G_n)$ is the list of generators.

Algorithm 7: ProveMultiple$_{\boldsymbol{G}}(x)$

Data: The list of generators $\boldsymbol{G} = (G_0, G_1, ..., G_n) \in E(\mathbb{F}_p)^{n+1}$
The private key $x \in \mathbb{Z}_q$
$k \in_{\mathrm{R}} \mathbb{Z}_q$
$h \leftarrow \mathcal{H}(Y_1||...||Y_n)$ // $Y_i = [x]G_i$, $i \in \{1, ..., n\}$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $z_i \leftarrow \mathcal{H}(i||h)$
end
$K \leftarrow [k](G_0 + \sum_{i=1}^{n}[z_i]G_i)$
$c \leftarrow \mathcal{H}(G_0||...||G_n||K||Y_0||...||Y_n)$
$r \leftarrow k + c \cdot x \pmod q$
$PK \leftarrow (K, c, r)$
return PK // $PK \in E(\mathbb{F}_p) \times \mathbb{Z}_q \times \mathbb{Z}_q$

The verifier accepts the proof if the algorithm **Verify**$_{\boldsymbol{G}}(PK, \boldsymbol{Y})$ returns true. The verification algorithm is described in algorithm 8.

Algorithm 8: VerifyMultiple$_{\boldsymbol{G}}(PK, \boldsymbol{Y})$

Data: The list of generators $\boldsymbol{G} = (G_0, G_1, ..., G_n) \in E(\mathbb{F}_p)^{n+1}$
The proof $PK = (K, c, r) \in E(\mathbb{F}_p) \times \mathbb{Z}_q \times \mathbb{Z}_q$
The list of public keys $\boldsymbol{Y} = (Y_0, Y_1, ..., Y_n) \in E(\mathbb{F}_p)^{n+1}$
$h \leftarrow \mathcal{H}(Y_1||...||Y_n)$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $z_i \leftarrow \mathcal{H}(i||h)$
end
if $c = \mathcal{H}(G_0||...||G_n||K||Y_0||...||Y_n)$
and $[r](G_0 + \sum_{i=1}^{n}([z_i]G_i)) = K + [c](Y_0 + \sum_{i=1}^{n}([z_i]Y_i))$ **then**
  $b \leftarrow 1$ // proof is valid
else
  $b \leftarrow 0$ // proof is invalid
end
return b // $b \in \mathbb{B}$

2.3 Hash Functions

A *cryptographic hash function* is an algorithm used for mapping data of arbitrary size to data of fixed size, also called the *hash value*. We define the hash function $\mathcal{H} : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$, where $\mathbb{B}^\ell$ represents a bit array of length ℓ. In practice, hash algorithms work on byte arrays instead of bit arrays. Thus, the length of the input or output array is $\ell/8$.

The hash value of any kind of data can be computed, for example a string, a number, even an object with a complex structure. The hash value would be the result of the hash function applied on the byte representation of that particular input data. A hash value can be computed of an arbitrary number of inputs at the same time. In such case, the hash function is applied on the concatenation of all byte representation of each input.

A hash function is known as a *one way function*, that means, one can easily verify that some input data maps to a given hash value but, if the input data is unknown, it is infeasible to calculate it given only a hash value. Another property of a cryptographic hash function is *collision resistance*. That means, it is difficult to find two different input data with the same hash values.

In our system, we will use the hash function called *SHA-256* that outputs bit arrays of 256 bits in length (32 byte array).

2.4 Elgamal Cryptosystem

The *Elgamal cryptosystem* is an asymmetric, randomized encryption scheme, where anybody can encrypt a message using the encryption key resulting in a *cryptogram*, while only the one in possession of the decryption key can extract the message of a cryptogram. The scheme consists of a triple (**KeyGen**, **Enc**, **Dec**) of algorithms that work on the elliptic curve described in section 2.1.4. The scheme is considered secure under the *discrete logarithm assumption*.

An Elgamal key pair is a tuple $(x, Y) \leftarrow \mathbf{KeyGen}()$ (algorithm 9), where x is a randomly chosen scalar representing the private decryption key and Y is an elliptic curve point corresponding to the public encryption key.

Algorithm 9: KeyGen()

$x \in_{\mathrm{R}} \mathbb{Z}_q$
$Y \leftarrow [x]G$
return (x, Y) // $(x, Y) \in \mathbb{Z}_q \times E(\mathbb{F}_p)$

The encryption algorithm $e = (R, C) \leftarrow \mathbf{Enc}_Y(M, r)$ (algorithm 10) can be used by anybody in possession of the encryption key Y, to generate a cryptogram on a message M, using the randomnizer r. The cryptogram e can be decrypted back to the original message M, only by the one in possession of the secret decryption key x, using the decryption algorithm $M \leftarrow \mathbf{Dec}_x(e)$ (algorithm 11). For convenience of notation, we define $\mathbb{E} = E(\mathbb{F}_p) \times E(\mathbb{F}_p)$ as the set of all possible cryptograms.

Algorithm 10: Enc$_Y(M, r)$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The message $M \in E(\mathbb{F}_p)$
The randomizer $r \in \mathbb{Z}_q$
$R \leftarrow [r]G$
$S \leftarrow [r]Y$
$C \leftarrow S + M$
$e \leftarrow (R, C)$
return e // $e \in \mathbb{E}$

Algorithm 11: $\mathbf{Dec}_x(e)$

Data: The decryption key $x \in \mathbb{Z}_q$
The cryptogram $e = (R, C) \in \mathbb{E}$
$S \leftarrow [x]R$
$M \leftarrow C - S$
return M // $M \in E(\mathbb{F}_p)$

Special encryption and decryption algorithms exist in case the message to be encrypted is not an elliptic curve point but a scalar. An extra step is required for mapping a curve point into a number from $\mathbb{Z}_q$, i.e. hashing the point S to get the integer $s \in \mathbb{Z}_q$. Formally, the message to be encrypted is $m \in \mathbb{Z}_q$. The algorithm for encrypting a scalar $e = (R, c) \leftarrow \mathbf{EncScalar}_Y(m, r)$ is described in algorithm 12, while the decryption algorithm $m \leftarrow \mathbf{DecScalar}_x(e)$ is described in algorithm 13.

Algorithm 12: $\mathbf{EncScalar}_Y(m, r)$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The message $m \in \mathbb{Z}_q$
The randomizer $r \in \mathbb{Z}_q$
$R \leftarrow [r]G$
$S \leftarrow [r]Y$
$s \leftarrow \mathcal{H}(S)$
$c \leftarrow s \cdot m \pmod q$
$e \leftarrow (R, c)$
return e // $e \in E(\mathbb{F}_p) \times \mathbb{Z}_q$

Algorithm 13: $\mathbf{DecScalar}_x(e)$

Data: The decryption key $x \in \mathbb{Z}_q$
The cryptogram $e = (R, c) \in E(\mathbb{F}_p) \times \mathbb{Z}_q$
$S \leftarrow [x]R$
$s \leftarrow \mathcal{H}(S)$
$m \leftarrow c \cdot s^{-1} \pmod q$
return m // $m \in \mathbb{Z}_q$

2.4.1 Proving the Content of a Cryptogram

Once a cryptogram is generated $e = (R, C) \leftarrow \mathbf{Enc}_Y(M, r)$, only the *sender* (the one who generated the cryptogram) and the *receiver* (the one in possession of the decryption key x) know the value of the message M. Both of them have the possibility to prove to somebody else (or publicly prove) the content of the cryptogram.

The one who generated the cryptogram can prove to a verifier that the cryptogram e contains message M by engaging in the protocol from figure 2 in order to prove the knowledge of the randomizer $PK[(r) : R = [r]G \wedge (C - M) = [r]Y]$. To generate a publicly verifiable proof, the *sender* can generate a non-interactive proof $PK \leftarrow \mathbf{ProveEquality}_{G,Y}(r)$ (algorithm 5). Any public

verifier is convinced that cryptogram e contains message M if the verification algorithm succeeds $\mathbf{VerifyEquality}_{G,Y}(PK, R, C - M)$ (algorithm 6).

At the same time, the one in possession of the decryption key x can prove the content of the cryptogram e to a verifier by engaging in the same protocol from figure 2 but this time for proving the knowledge of the decryption key $PK[(x) : Y = [x]G \wedge (C - M) = [x]R]$. To generate a publicly verifiable proof, the *receiver* of the cryptogram can generate a non-interactive proof $PK \leftarrow \mathbf{ProveEquality}_{G,R}(x)$ (algorithm 5). Any public verifier is convinced that cryptogram e contains message M if the verification algorithm succeeds $\mathbf{VerifyEquality}_{G,R}(PK, Y, C - M)$ (algorithm 6).

2.4.2 Homomorphic Encryption

Elgamal point encryption based on elliptic curve cryptographic primitive is a *homomorphic* encryption scheme with respect to point addition. That means, component wise addition of two cryptograms would result in a new, valid cryptogram that contains the two messages added together.

$$\mathbf{Enc}_Y(M_1, r_1) + \mathbf{Enc}_Y(M_2, r_2) = \mathbf{Enc}_Y(M_1 + M_2, r_1 + r_2)$$

The resulting encryption of the homomorphic addition of two cryptograms is $e' = (R', C') \leftarrow \mathbf{AddEnc}(e_1, e_2)$ (algorithm 14).

Algorithm 14: AddEnc(e_1, e_2)

Data: The first cryptogram $e_1 = (R_1, C_1) \in \mathbb{E}$
The second cryptogram $e_2 = (R_2, C_2) \in \mathbb{E}$
$R' \leftarrow R_1 + R_2$
$C' \leftarrow C_1 + C_2$
$e' \leftarrow (R', C')$
return e' // $e' \in \mathbb{E}$

Following the procedure above, we can *re-encrypt* a given encryption $e = (R, C) \leftarrow \mathbf{Enc}_Y(M, r)$ by homomorphically adding it to an *empty cryptogram* (an encryption of the neutral point $\mathcal{O}$) with randomizer $r' \in_{\mathrm{R}} \mathbb{Z}_q$. The result is a new, randomly different cryptogram that contains the same message M. The process of generating the new cryptogram $e' = (R', C') \leftarrow \mathbf{ReEnc}_Y(e, r')$ is described in algorithm 15.

Algorithm 15: ReEnc$_Y(e, r')$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The initial cryptogram $e = (R, C) \in \mathbb{E}$
The new randomizer $r' \in \mathbb{Z}_q$
$e_2 \leftarrow \mathbf{Enc}_Y(\mathcal{O}, r')$ // algorithm 10
$e' \leftarrow \mathbf{AddEnc}(e, e_2)$ // algorithm 14
return e' // $e' \in \mathbb{E}$

Usually, a re-encrypted cryptogram comes with a re-encryption proof to assure that the content of the cryptogram has not been changed. The proof is

a non-interactive discrete logarithm equality proof (described in section 2.2.2) $PK = (\boldsymbol{K}, c, r) \leftarrow \mathbf{Prove}_{G,Y}(r')$, while the proof verification algorithm is $b \leftarrow \mathbf{Verify}_{G,Y}(PK, R' - R, C' - C)$, where $b \in \mathbb{B}$.

2.4.3 Elgamal Threshold Cryptosystem

A t out of n threshold cryptosystem is an encryption scheme where the decryption key is split among n key holders, called *trustees*. Anybody can encrypt a message using the encryption key. Decryption of a message happens during a process in which at least t trustees have to collaborate in a cryptographic protocol. It is recommended that $t \geq 2/3 \cdot n$. The entire scheme is inspired from [5] which is based on mathematical principles of the threshold cryptosystem [6, 7].

The key generation algorithm outputs $(sx_1, ..., sx_n, Y) \leftarrow \mathbf{KeyGen}(n, t)$, where Y is the public encryption key and each sx_i is a private share of the decryption key, one for each of the n trustees. The algorithm spans over the protocol described in figure 4, which we call *the threshold ceremony*.

During the *threshold ceremony*, all trustees generate a private-public key pair (x_i, Y_i) and publish to the server their public keys. The public encryption key is computed by the sum of the public keys of all trustees $Y = \sum_{i=1}^{n} Y_i$, while nobody being in the possession of the decryption key $x = \sum_{i=1}^{n} x_i$, because all x_i are secret. Instead, all trustees work together to distribute x such that any t trustees can find it when necessary.

Each trustee $\mathcal{T}_i$ generates a polynomial function of degree $t - 1$

$$f_i(z) = x_i + p_{i,1} \cdot z + ... + p_{i,t-1} \cdot z^{t-1}$$

, where $p_{i,k} \in_{\mathrm{R}} \mathbb{Z}_q$ with $k \in \{1, ..., t-1\}$. Next, all trustees publish to the server the curve points $(P_{i,1}, ..., P_{i,t-1})$, where each $P_{i,k} \leftarrow [p_{i,k}]G$. We call $(p_{i,k}, P_{i,k})$ a private-public polynomial coefficient pair.

When all trustees have published their public coefficients, each trustee computes a *partial secret share of the decryption key* for each of the other trustees by $s_{i,j} \leftarrow f_i(j)$, and encrypts them with that specific trustee's public key $e_{i,j} \leftarrow \mathbf{EncScalar}_{Y_j}(s_{i,j})$. Finally, all trustees publish to the server all encrypted *partial secret shares of the decryption key*.

By encrypting the partial secret shares with different trustee's public key, we make sure that only that specific trustee can read his *partial secret shares of the decryption key*. This procedure is a small deviation from [5], which we introduced in order to simulate a secret communication channel between each two trustees.

Finally, each trustee $\mathcal{T}_i$ downloads his encrypted *partial secret shares* $e_{j,i}$, with $j \in \{1, ..., n\}$, decrypts them $s_{j,i} \leftarrow \mathbf{DecScalar}_{x_i}(e_{j,i})$ and validates that they are consistent with the polynomial coefficients of the other trustees $[s_{j,i}]G \stackrel{?}{=} Y_j + \sum_{k=1}^{t-1} [i^k] P_{j,k}$. If all *partial secret shares* validate, then trustee $\mathcal{T}_i$ computes his *secret share of the decryption key* by $sx_i \leftarrow \sum_{j=1}^{n} s_{j,i}$.

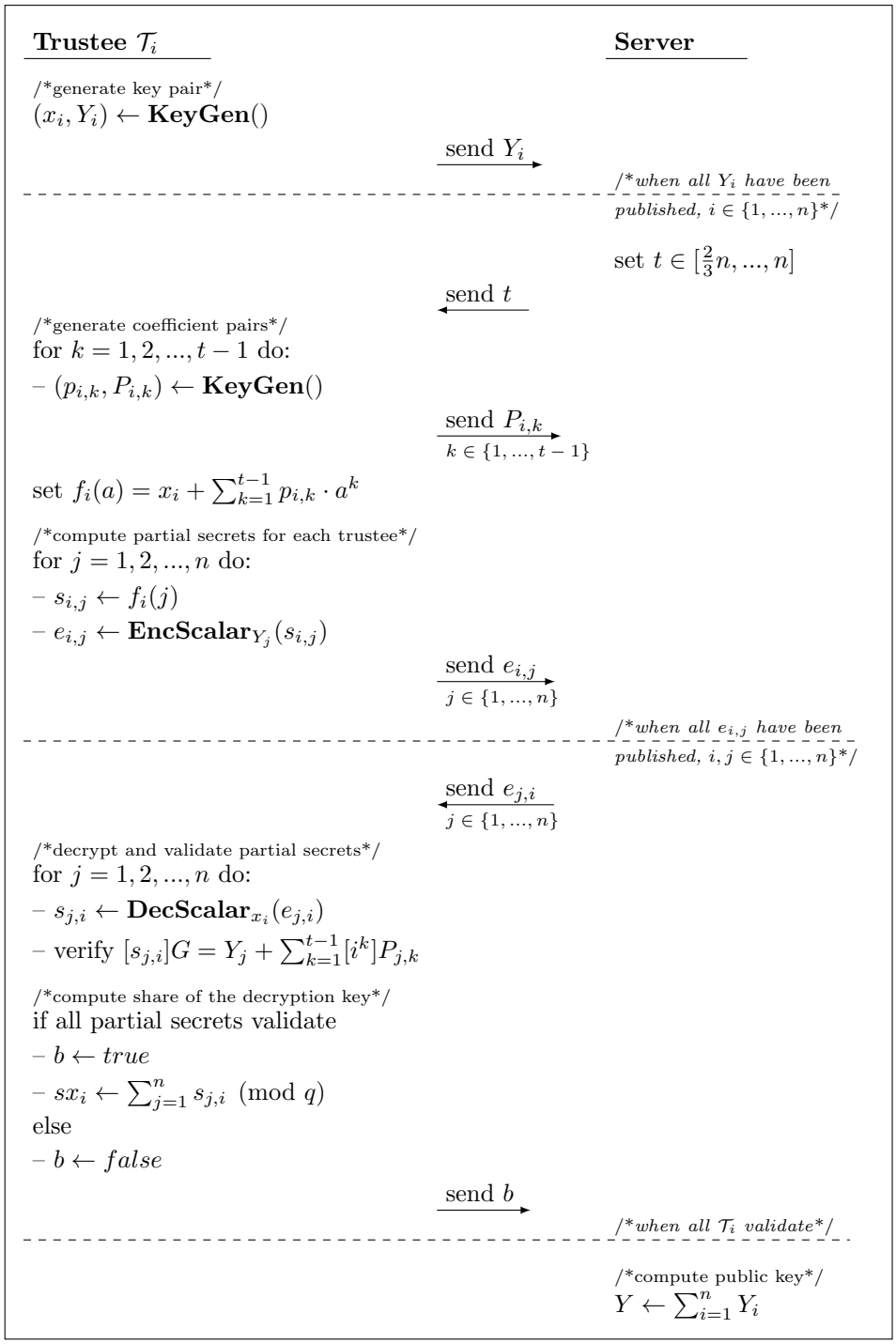

Figure 4: Threshold ceremony

At the end of the *threshold ceremony*, for each trustee $\mathcal{T}_i$, with $i \in \{1, ..., n\}$, the *public share of the decryption key* ($sY_i = [sx_i]G$) is publicly computable by the following:

$$sY_i \leftarrow \sum_{j=1}^{n}(Y_j + \sum_{k=1}^{t-1}[i^k]P_{j,k})$$

The encryption algorithm of the threshold cryptosystem is identical to the *Point Encryption* algorithm described in section 2.4. Note that using the threshold encryption scheme, we can only encrypt a message M that is represented as a point on the elliptic curve

$$e = (R, C) \leftarrow \mathbf{Enc}_Y(M, r).$$

The decryption algorithm of the threshold cryptosystem is inspired from paper [6]. At least t trustees are needed to collaborate in the protocol described in figure 5 in order to extract the message of a cryptogram $M \leftarrow \mathbf{Dec}_T(e)$, where $T \subset \{1, ..., n\}$ is the subset of trustees that do participate in the decryption protocol, with $|T| \geq t$ and $e = (R, C)$ is the cryptogram to be decrypted. Each trustee $\mathcal{T}_i$, with $i \in T$, computes a partial decryption $S_i \leftarrow [sx_i]G$ and sends it to the server, where sx_i is trustee's share of the decryption key. The trustee also publishes a proof of correct decryption in form of a non-interactive discrete logarithm zero knowledge proof $PK \leftarrow \mathbf{ProveEquality}_{G,R}(sx_i)$ (algorithm 5).

When receiving a partial decryption from a trustee $\mathcal{T}_i$, the server accepts it if the proof of correct decryption validates by $\mathbf{VerifyEquality}_{G,R}(PK, sY_i, S_i)$ (algorithm 6), where sY_i is trustee's *public share of the decryption key.*

After it received valid, partial decryptions from all trustees $\mathcal{T}_i$, with $i \in T$, the server aggregates all partial decryptions together to finalize the decryption and to output the message M. The aggregation process from [6] is described as follows:

Basically, $M = C - [x]R$, where x is the main decryption key that nobody has. A possible way of computing $[x]R$ is by calculating the *Lagrange Interpolation Polynomial* where each term is a partial decryption received from a trustee S_i that needs to be multiplied by the *Lagrange Interpolation Polynomial coefficient* which is $\lambda(i) = \prod_{j \in T, j \neq i} \frac{-j}{i-j} \pmod q$. Formally, $M \leftarrow C - \sum_{i \in T}[\lambda(i)]S_i$.

Note that the *Lagrange Interpolation Polynomial* can be computed only when the number of terms is at least the degree of the polynomial, i.e $|T| \geq t$.

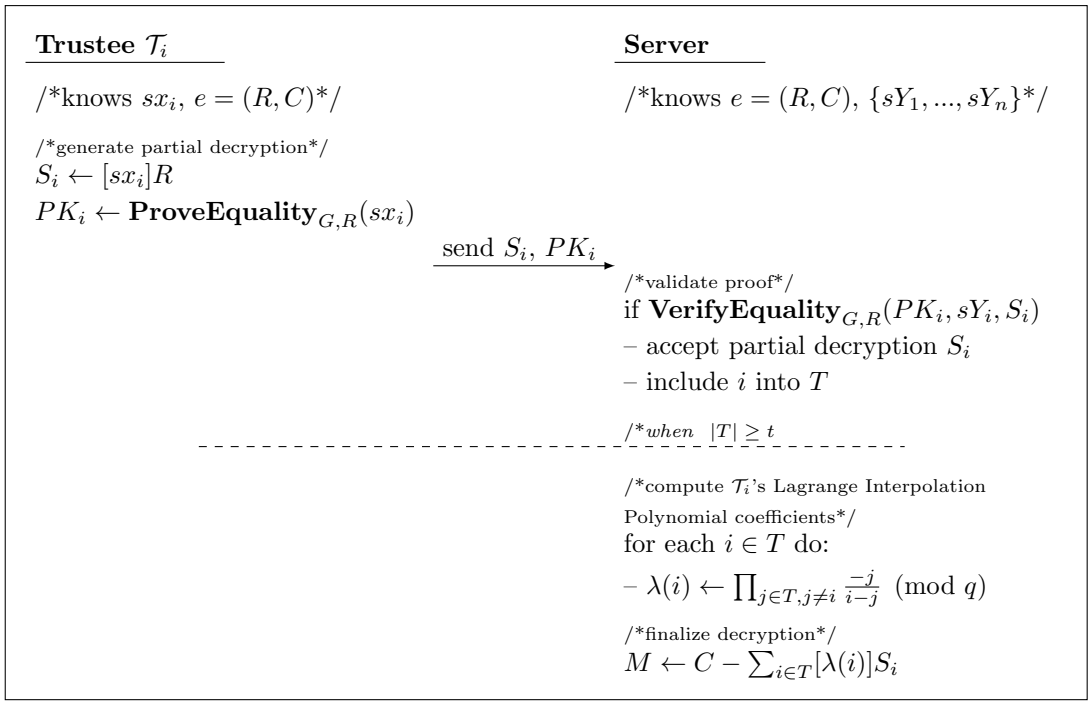

Figure 5: Threshold decryption

2.5 Schnorr Digital Signature

The *Schnorr digital signature scheme*, introduced in [8], consists of a triple of algorithms (**KeyGen**, **Sign**, **VerifySignature**), which are based on elliptic curve cryptographic primitive.

A Schnorr key pair is a tuple $(x, Y) \leftarrow \textbf{KeyGen}()$ (algorithm 9), where x is the random, private signing key and Y is the corresponding public signature verification key.

Only the owner of the signing key is able to generate a signature $\sigma = (c, s) \leftarrow \textbf{Sign}_x(m)$, on an arbitrary message $m \in \mathbb{B}^*$. In order to generate a signature, the signer follows algorithm 16.

Algorithm 16: $\textbf{Sign}_x(m)$

Data: The signing key $x \in \mathbb{Z}_q$
The message to be signed $m \in \mathbb{B}^*$
$r \in_{\mathrm{R}} \mathbb{Z}_q$
$K \leftarrow [r]G$
$c \leftarrow \mathcal{H}(K||m)$
$s \leftarrow r - c \cdot x \pmod q$
$\sigma \leftarrow (c, s)$
return σ // $\sigma \in \mathbb{Z}_q \times \mathbb{Z}_q$

Given a signature σ on a message m, anybody in the possession of the public verification key Y is able to verify the validity of the signature $b \leftarrow \textbf{VerifySignature}_Y(\sigma, m)$, with $b \in \mathbb{B}$ which represents $true$ or $false$. The signature verification algorithm is described in algorithm 17

Algorithm 17: VerifySignature$_Y(\sigma, m)$

Data: The verification key $Y \in E(\mathbb{F}_p)$
The signature $\sigma = (c, s) \in \mathbb{Z}_q \times \mathbb{Z}_q$
The signed message $m \in \mathbb{B}^*$
$K \leftarrow [s]G + [c]Y$
if $c = \mathcal{H}(K||m)$ **then**
  $b \leftarrow 1$ // signature is valid
else
  $b \leftarrow 0$ // signature is invalid
end
return b // $b \in \mathbb{B}$

2.6 Furukawa's Proof of Shuffle

A *cryptographic shuffle* is a process that, given as input a list of cryptograms, outputs another list of cryptograms such that each cryptogram from the input list is re-encrypted and permuted in a random new order, forming the output list.

Formally, given a list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$, with $e_i = (R_i, C_i)$ and $i \in \{1, ..., n\}$, a list of randomizers $\boldsymbol{r'} = (r'_1, ..., r'_n) \in \mathbb{Z}_q^n$ and a permutation $\psi : \{1, ..., n\} \rightarrow \{1, ..., n\}$ from the set Ψ_n of all permutations of n elements, the shuffle algorithm outputs the list $\boldsymbol{e'} = (e'_1, ..., e'_n) \leftarrow \textbf{Shuffle}_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi)$ (algorithm 18) where each $e'_i = (R'_i, C'_i) \leftarrow \textbf{ReEnc}_Y(e_j, r'_i)$ for $j = \psi(i)$.

Algorithm 18: Shuffle$_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi)$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
The list of randomizers $\boldsymbol{r'} = (r'_1, ..., r'_n) \in \mathbb{Z}_q^n$
The permutation $\psi \in \Psi_n$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $e'_i \leftarrow \textbf{ReEnc}_Y(e_{\psi(i)}, r'_i)$ // algorithm 15
end
$\boldsymbol{e'} \leftarrow (e'_1, ..., e'_n)$
return $\boldsymbol{e'}$ // $\boldsymbol{e'} \in \mathbb{E}^n$

The really interesting aspect of a shuffle is how to prove in zero knowledge that the shuffling calculations were done correctly and that no content of the cryptograms has been changed. Our proof of shuffle is based on an algorithm presented by Jun Furukawa in [9] (later updated in [10, 11]). We adapted all calculations to elliptic curve point operations. The proof consists of two parts:

1. proving that no content of the cryptograms has been changed, i.e. same randomizer r'_i and permutation ψ have been used to re-encrypt both parts

of the cryptogram tuples

$$e'_i = (R'_i, C'_i) = (R_{\psi(i)} + [r'_i]G, C_{\psi(i)} + [r'_i]Y)$$

2. proving that ψ is indeed a permutation, which is equivalent to proving that the matrix $\boldsymbol{A}_\psi$ is a permutation matrix, where

$$\boldsymbol{A}_\psi = \begin{pmatrix} A_{1,1} & \dots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{n,1} & \dots & A_{n,n} \end{pmatrix} \in \{0,1\}^{n \cdot n} \text{ , where } A_{i,j} = \begin{cases} 1, & \text{if } \psi(i) = j \\ 0, & \text{otherwise} \end{cases}$$

The shuffle proof $PK = (R', C', U, \boldsymbol{U}, T, \dot{\boldsymbol{T}}, V, \dot{\boldsymbol{V}}, \dot{V}, W, \dot{\boldsymbol{W}}, \dot{W}, \boldsymbol{c}, z, \boldsymbol{s}, \lambda')$ is constructed by the protocol presented in figure 6. In our proof construction, $(R', C', U, \boldsymbol{U}, T, \dot{\boldsymbol{T}}, V, \dot{\boldsymbol{V}}, \dot{V}, W, \dot{\boldsymbol{W}}, \dot{W})$ represent the commitment, $\boldsymbol{c}$ represents the challenge and $(z, \boldsymbol{s}, \lambda')$ is the response of the proof. The verifier accepts the proof in the end (is convinced that the contents of the cryptograms have not changed during the shuffling process) if the final verification step validates.

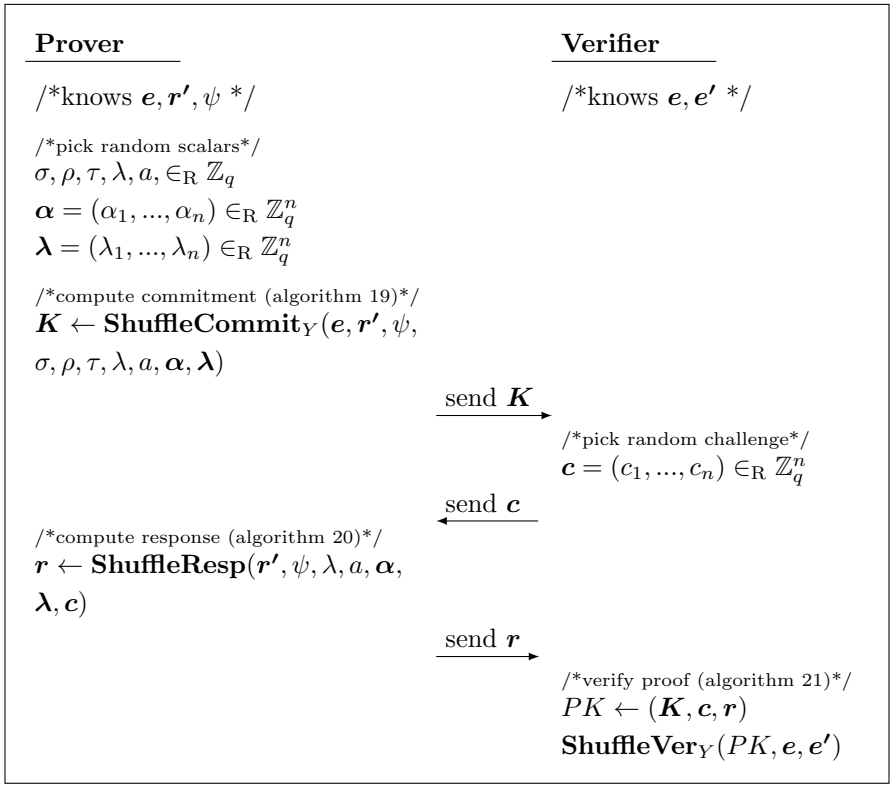

Figure 6: Protocol for proving a shuffle

For notation convenience, when we express the shuffling of a list, we will use the permutation notation, as in [10], instead of the permutation matrix. For example, given a list of elements $(g_1, ..., g_n)$ and a permutation $\psi \in_R \Psi_n$, having its permutation matrix $\boldsymbol{A}_\psi = (A_{1,1}, ..., A_{n,n}) \in \{0,1\}^{n \cdot n}$, we express a shuffled element like $g'_i = g_{\psi(i)}$, instead of $g'_i = \sum_{j=1}^{n} g_j \cdot A_{i,j}$.

For convenience, the entire protocol has been split into multiple sub-algorithms (i.e. **ShuffleCommit**, **ShuffleResp**, **ShuffleVer**) that help in constructing the proof. Changes in variable names, compared to [9] are stated in appendix A.1.

In section "Constructing the Main Protocol" of [9], it is presented an extra security mechanism, needed in case the prover knows the randomizers used in the generation of the initial list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n)$. New basis $\{\tilde{G}, \tilde{G}_1, ..., \tilde{G}_n\}$ must be set independently chosen from $\boldsymbol{e}$. Next, it must be proven that the same randomizers r'_i and permutation ψ has been applied to the basis $\{\tilde{G}, \tilde{G}_1, ..., \tilde{G}_n\}$ and to both tuples in each cryptogram e_i.

$$(\tilde{G}'_i, R'_i, C'_i) = (\tilde{G}_{\psi(i)} + [r'_i]\tilde{G}, R_{\psi(i)} + [r'_i]G, C_{\psi(i)} + [r'_i]Y)$$

We can skip the computation of $\tilde{G}$, $\tilde{G}_i$ and $\tilde{G}'_i$, while our shuffle proof still has the same security properties as in [9], on the assumption that the prover is not in the possession of the randomizers associated to each cryptogram of $\boldsymbol{e}$.

Algorithm 19: ShuffleCommit$_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi, \sigma, \rho, \tau, \lambda, a, \boldsymbol{\alpha}, \boldsymbol{\lambda})$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
$e_i = (R_i, C_i)$, where $i \in \{1, ..., n\}$
The list of randomizers $\boldsymbol{r'} = (r'_1, ..., r'_n) \in \mathbb{Z}_q^n$
The permutation $\psi \in \Psi_n$
Scalars $\sigma, \rho, \tau, \lambda, a, \in \mathbb{Z}_q$ and
$\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_n), \boldsymbol{\lambda} = (\lambda_1, ..., \lambda_n) \in \mathbb{Z}_q^n$

`/* commitment to cryptogram list e */`

$R' \leftarrow [a]G + \sum_{i=1}^{n} [\alpha_i] R_i$
$C' \leftarrow [a]Y + \sum_{i=1}^{n} [\alpha_i] C_i$

`/* compute scalars */`

for $i \leftarrow 1$ **to** n **by** 1 **do**
  $\dot{t}_i \leftarrow \tau \cdot \lambda_i + 3 \cdot \alpha_{\psi(i)} \pmod q$
  $\dot{v}_i \leftarrow \rho \cdot r'_i + 3 \cdot \alpha^2_{\psi(i)} \pmod q$
  $\dot{w}_i \leftarrow \sigma \cdot r'_i + 2 \cdot \alpha_{\psi(i)} \pmod q$
end
$\dot{v} \leftarrow \tau \cdot \lambda + \rho \cdot a + \sum_{i=1}^{n} \alpha_i^3 \pmod q$
$\dot{w} \leftarrow \sigma \cdot a + \sum_{i=1}^{n} \alpha_i^2 \pmod q$

`/* compute commitment to permutation */`

$U \leftarrow [\lambda]G$, $T \leftarrow [\tau]G$, $V \leftarrow [\rho]G$
$\dot{V} \leftarrow [\dot{v}]G$, $W \leftarrow [\sigma]G$, $\dot{W} \leftarrow [\dot{w}]G$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $U_i \leftarrow [\lambda_i]G$, $\dot{T}_i \leftarrow [\dot{t}_i]G$, $\dot{V}_i \leftarrow [\dot{v}_i]G$, $\dot{W}_i \leftarrow [\dot{w}_i]G$
end
$\boldsymbol{U} \leftarrow (U_1, ..., U_n)$, $\boldsymbol{\dot{T}} \leftarrow (\dot{T}_1, ..., \dot{T}_n)$
$\boldsymbol{\dot{V}} \leftarrow (\dot{V}_1, ..., \dot{V}_n)$, $\boldsymbol{\dot{W}} \leftarrow (\dot{W}_1, ..., \dot{W}_n)$
$\boldsymbol{K} \leftarrow (R', C', U, \boldsymbol{U}, T, \boldsymbol{\dot{T}}, V, \boldsymbol{\dot{V}}, \dot{V}, W, \boldsymbol{\dot{W}}, \dot{W})$
return $\boldsymbol{K}$ `//` $\boldsymbol{K} \in E(\mathbb{F}_p)^{8+4 \cdot n}$

Algorithm 20: ShuffleResp$(\boldsymbol{r'}, \psi, \lambda, a, \boldsymbol{\alpha}, \boldsymbol{\lambda}, \boldsymbol{c})$

Data: The list of randomizers $\boldsymbol{r'} = (r'_1, ..., r'_n) \in \mathbb{Z}_q^n$
The permutation $\psi \in \Psi_n$
Scalars $\lambda, a \in \mathbb{Z}_q$ and $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_n), \boldsymbol{\lambda} = (\lambda_1, ..., \lambda_n) \in \mathbb{Z}_q^n$
The list of challenges $\boldsymbol{c} = (c_1, ..., c_n) \in \mathbb{Z}_q^n$

$\lambda' \leftarrow \sum_{i=1}^{n} \lambda_i \cdot c_i^2 + \lambda \pmod q$
$z \leftarrow \sum_{i=1}^{n} r'_i \cdot c_i + a \pmod q$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $s_i \leftarrow \alpha_i + c_{\psi^{-1}(i)} \pmod q$
end
$\boldsymbol{s} \leftarrow (s_1, ..., s_n)$
return $(\lambda', z, \boldsymbol{s})$ // $(\lambda', z, \boldsymbol{s}) \in \mathbb{Z}_q^{2+n}$

Algorithm 21: ShuffleVer$_Y(PK, \boldsymbol{e}, \boldsymbol{e'})$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The proof of shuffle $PK = (\boldsymbol{K}, \boldsymbol{c}, \boldsymbol{r})$
$\boldsymbol{K} = (R', C', U, \boldsymbol{U}, T, \dot{\boldsymbol{T}}, V, \dot{\boldsymbol{V}}, \dot{V}, W, \dot{\boldsymbol{W}}, \dot{W}) \in E(\mathbb{F}_p)^{8+4\cdot n}$
$\boldsymbol{U} = (U_1, ..., U_n), \dot{\boldsymbol{T}} = (\dot{T}_1, ..., \dot{T}_n) \in E(\mathbb{F}_p)^n$
$\dot{\boldsymbol{V}} = (\dot{V}_1, ..., \dot{V}_n), \dot{\boldsymbol{W}} = (\dot{W}_1, ..., \dot{W}_n) \in E(\mathbb{F}_p)^n$
$\boldsymbol{c} = (c_1, ..., c_n) \in \mathbb{Z}_q^n$
$\boldsymbol{r} = (\lambda', z, \boldsymbol{s}) \in \mathbb{Z}_q^{2+n}$
$\boldsymbol{s} = (s_1, ..., s_n) \in \mathbb{Z}_q^n$
The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
$e_i = (R_i, C_i)$, with $i \in \{1, ..., n\}$
The output list of cryptograms $\boldsymbol{e'} = (e'_1, ..., e'_n) \in \mathbb{E}^n$
$e'_i = (R'_i, C'_i)$, with $i \in \{1, ..., n\}$

if $[\lambda']G = U + \sum_{i=1}^{n} [c_i^2] U_i$
and $[\lambda']T + [z]V + [\sum_{i=1}^{n} s_i^3 - c_i^3]G = \dot{V} + \sum_{i=1}^{n} [c_i]\dot{V}_i + [c_i^2]\dot{T}_i$
and $[z]W + [\sum_{i=1}^{n} s_i^2 - c_i^2]G = \dot{W} + \sum_{i=1}^{n} [c_i]\dot{W}_i$
and $[z]G + \sum_{i=1}^{n} [s_i]R_i = R' + \sum_{i=1}^{n} [c_i]R'_i$
and $[z]Y + \sum_{i=1}^{n} [s_i]C_i = C' + \sum_{i=1}^{n} [c_i]C'_i$
then
  $b \leftarrow 1$ `// proof is valid`
else
  $b \leftarrow 0$ `// proof is invalid`
end
return b // $b \in \mathbb{B}$

The proof of shuffle can be made non-interactive by having the list of challenges $\boldsymbol{c}$ calculated by a hash function that takes as input the initial list of cryptograms $\boldsymbol{e}$, the commitment of the proof $\boldsymbol{K}$ and the outputted list of cryptograms $\boldsymbol{e'}$, as presented in [10]. The algorithm for computing the challenges is described in algorithm 22.

Algorithm 22: ShuffleChall$(\boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{K})$

Data: The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
The output list of cryptograms $\boldsymbol{e'} = (e'_1, ..., e'_n) \in \mathbb{E}^n$
The commitment of the proof $\boldsymbol{K} \in E(\mathbb{F}_p)^{8+4 \cdot n}$

$h \leftarrow \mathcal{H}(e_1||...||e_n||\boldsymbol{K}||e'_1||...||e'_n)$
for $i \leftarrow 1$ **to** n **by** 1 **do**
  $c_i \leftarrow \mathcal{H}(i||h)$
end
$\boldsymbol{c} = (c_1, ..., c_n)$
return $\boldsymbol{c}$ // $c \in \mathbb{Z}_q^n$

Now the entire proof of shuffle $PK \leftarrow \textbf{ProveShuffle}_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi)$ (algorithm 23) can be generated by the prover alone by running the algorithm 23. To verify a non-interactive PK, any public verifier has to run the algorithm $b \leftarrow \textbf{VerifyShuffle}_Y\ (PK, \boldsymbol{e}, \boldsymbol{e'})$ (algorithm 24), where $b \in \mathbb{B}$ represents whether the proof is valid or not.

Algorithm 23: ProveShuffle$_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi)$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
The list of randomizers $\boldsymbol{r'} = (r'_1, ..., r'_n) \in \mathbb{Z}_q^n$
The permutation $\psi \in \Psi_n$

$\sigma, \rho, \tau, \lambda, a, \in_{\mathrm{R}} \mathbb{Z}_q$
$\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_n), \boldsymbol{\lambda} = (\lambda_1, ..., \lambda_n) \in_{\mathrm{R}} \mathbb{Z}_q^n$
$\boldsymbol{K} \leftarrow$ **ShuffleCommit**$_Y(\boldsymbol{e}, \boldsymbol{r'}, \psi, \sigma, \rho, \tau, \lambda, a, \boldsymbol{\alpha}, \boldsymbol{\lambda})$ `// commitment (alg. 19)`
$\boldsymbol{c} \leftarrow$ **ShuffleChall**$(\boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{K})$ `// challenge(alg. 22)`
$\boldsymbol{r} \leftarrow$ **ShuffleResp**$(\boldsymbol{r'}, \psi, \lambda, a, \boldsymbol{\alpha}, \boldsymbol{\lambda}, \boldsymbol{c})$ `// response(alg. 20)`
$PK \leftarrow (\boldsymbol{K}, \boldsymbol{c}, \boldsymbol{r})$
return PK `//` $PK \in E(\mathbb{F}_p)^{8+4\cdot n} \times \mathbb{Z}_q^n \times \mathbb{Z}_q^{2+n}$

Algorithm 24: VerifyShuffle$_Y(PK, \boldsymbol{e}, \boldsymbol{e'})$

Data: The encryption key $Y \in E(\mathbb{F}_p)$
The proof of shuffle $PK = (\boldsymbol{K}, \boldsymbol{c}, \boldsymbol{r}) \in E(\mathbb{F}_p)^{8+4\cdot n} \times \mathbb{Z}_q^n \times \mathbb{Z}_q^{2+n}$
The input list of cryptograms $\boldsymbol{e} = (e_1, ..., e_n) \in \mathbb{E}^n$
The output list of cryptograms $\boldsymbol{e'} = (e'_1, ..., e'_n) \in \mathbb{E}^n$

if $\boldsymbol{c} =$ **ShuffleChall**$(\boldsymbol{e}, \boldsymbol{e'}, \boldsymbol{K})$ `// algorithm 22`
and ShuffleVer$_Y(PK, \boldsymbol{e}, \boldsymbol{e'})$ `// algorithm 21`
then
  $b \leftarrow 1$ `// proof is valid`
else
  $b \leftarrow 0$ `// proof is invalid`
end
return b `//` $b \in \mathbb{B}$

3 Election Protocol

3.1 Involved Parties

In our election process multiple parties are involved. Each party represents a human with access to a computer or simply a process/software that follows a particular protocol. All these parties can be categorized into the following 6 types:

- *Election Administrator* $\mathcal{E}$: There exists one or many administrators that are responsible for setting up the election event. An administrator is a person controlling a computer with access to internet.
- *Voter* $\mathcal{V}$: There exists a list of eligible voters, each noted $\mathcal{V}_i$, with $i \in \{1, ..., n_{\mathrm{v}}\}$, where n_{v} is the total number of voters. A voter is a human being that is allowed to participate in this election. A voter needs to have access to a computer that has an internet connection. Voters are the ones to generate vote cryptograms.
- *Printing Authority* $\mathcal{P}$: There is a set of printing authorities, each noted as $\mathcal{P}_i$, with $i \in \{1, ..., n_{\mathrm{p}}\}$, where n_{p} is the total number of printing authorities. Each of them is responsible for generating voter credentials and distribute them privately to the voters. It is recommended that each printing authority should use a different communication channel for distributing credentials (e.g. e-mail, post, SMS). A printing authority is an institution (consisting of humans and software processes) that has to follow our protocol.
- *Trustee* $\mathcal{T}$: There is a set of trustees, each noted as $\mathcal{T}_i$, with $i \in \{1, ..., n_{\mathrm{t}}\}$, where n_{t} is the total number of trustees. A trustee is a human controlling a computer that has the *Trustee Application* installed. Trustees are responsible for preserving the privacy and the fairness of the election during the election phase by working together to build the election encryption key while safely storing their shares of the decryption key.
- *Mix Node* $\mathcal{M}$: There exists a mixnet represented by a set of mix nodes, each noted as $\mathcal{M}_i$, with $i \in \{1, ..., n_{\mathrm{m}}\}$, where n_{m} is the total number of mix nodes. A mix node is a software process that runs on a separate server with enough computation capability. The mixnet is responsible for preserving the anonymity of the election by shuffling the entire list of vote cryptograms in an indistinguishable way.
- *Bulletin Board* $\mathcal{B}$: There is one bulletin board that contains all the information about an election. It is the communication central unit as all other parties push/pull data to/from it. It consists of three parts:
    1. *election configuration*, which has to be set up during the pre-election phase. All data included here is described at section 3.2.

2. *list of vote cryptograms*, which is being populated during the election phase. The list is implemented with the following properties: no elements of the list are ever removed or modified; each new element is appended at the end of the list.
3. *mixing and decryption files*, which are collected during the post-election phase. These files contribute in the process of computing the result of the election.

The bulletin board is a piece of software, accessible over internet by any other actor involved in the election process.

3.2 Pre-election Phase

During the *pre-election phase* the *election administrators* have to gather all the information needed to set up an election. That includes:

- the election start date and end date
- the election question and election type
- the possible answers (vote options) $\boldsymbol{m} = (m_1, ..., m_{n_c})$, where n_c is the number of vote options, each representing a candidate name
- the list of eligible voters $\boldsymbol{\mathcal{V}} = (\mathcal{V}_1, ..., \mathcal{V}_{n_v})$. Each voter is defined by contact information for each of the communication channels that is used in the voter credential distribution process (section 3.2.1), e.g. an e-mail address, a postal address or a phone number.
- the list of printing authorities $\boldsymbol{\mathcal{P}} = (\mathcal{P}_1, ..., \mathcal{P}_{n_p})$ used for distributing voter credentials. Each printing authority is specified to use a particular communication channel for distributing voter credentials, e.g. e-mail, post or SMS.
- the lists of trustees $\boldsymbol{\mathcal{T}} = (\mathcal{T}_1, ..., \mathcal{T}_{n_t})$ and mix nodes $\boldsymbol{\mathcal{M}} = (\mathcal{M}_1, ..., \mathcal{M}_{n_m})$.

Next, the bulletin board will generate its signing key pair $(x_{\text{sign}}, Y_{\text{sign}}) \leftarrow$ **KeyGen**() (algorithm 9), where x_{sign} is the signing key and will be kept secret throughout the election period, and Y_{sign} is the public signature verification key.

Next, the system maps all vote options $\boldsymbol{m}$ to unique points on the elliptic curve $\boldsymbol{M} = (M_1, ..., M_{n_c}) \in E(\mathbb{F}_p)^{n_c}$ with each $M_i =$ **String2Point**(m_i) (algorithm 1). All vote options have to be represented in $E(\mathbb{F}_p)$ so they can be used in cryptographic procedures. The mapping algorithm is described at section 3.2.2.

Next, all trustees participate in the *threshold ceremony*, described in section 2.4.3 in order to generate the election encryption key Y_{enc} and each trustee's share of the decryption key sx_i. Formally, $(sx_1, ..., sx_{n_t}, Y_{\text{enc}}) \leftarrow$ **KeyGen**(n_t, t) (figure 4), where t is the threshold limit for decryption.

Finally, all printing authorities participate in the *voter credential distribution process* to distribute voter credentials. Also, public signature verification keys are computed for all voters. The process is described in section 3.2.1.

3.2.1 Voter Credential Distribution Process

Each *Printing Authority* $\mathcal{P}_i \in \boldsymbol{\mathcal{P}}$, receives a list of voters consisting of contact details for each voter $\boldsymbol{a} = (a_1, ..., a_{n_\text{v}})$ in form of e-mail addresses or postal addresses or phone numbers, depending on the printing authority's communication channel. The printing authority generates a random key pair for each of them $(x_{i,j}, Y_{i,j}) \leftarrow \textbf{KeyGen}()$ (algorithm 9), with $j \in \{1, ..., n_\text{v}\}$. The Printing Authority distributes the secret key $x_{i,j}$ to that specific voter $\mathcal{V}_j$ (using voter's contact detail a_j) and appends the corresponding public key $Y_{i,j}$ in the list of voters next to the specific voter $\mathcal{V}_j$.

All printing authorities return to the *Bulletin Board* the lists with voters contact details and public keys $(a_j, Y_{i,j})$. The *Bulletin Board* combines all public keys received from all printing authorities for each voter to form voter's public signature verification key $Y_j = \sum_{i=1}^{n_\text{p}} Y_{i,j}$.

For authenticating to the voting system, the voter $\mathcal{V}_j \in \boldsymbol{\mathcal{V}}$ has to input in the browser all secret keys $(x_{1,j}, ..., x_{n_\text{p},j})$ received via different channels from all printing authorities. The browser will combine all of them to form the voter's signing key $x_j = \sum_{i=1}^{n_\text{p}} x_{i,j} \pmod q$.

3.2.2 Mapping Vote Options on the Elliptic Curve

An expressed vote (a vote in plain text) must be able to be converted, deterministically, into an elliptic curve point in order to be used in our cryptographic protocols. Additionally, a point from the elliptic curve must be able to be turned back to a plain text vote, if the point has been constructed from a plain text. Consequently, only a finite number of points from $E(\mathbb{F}_p)$ can be mapped to a plain text that can be interpreted as valid votes. All the other points will be regarded as invalid votes.

The mapping procedure from a plain text m to an elliptic curve point M is called $M \leftarrow \textbf{String2Point}(m)$ and is described in the algorithm 1. Recovering the text of the vote m from an elliptic curve point M can be done by calling $m \leftarrow \textbf{Point2String}(M)$ (algorithm 2).

Depending on the election type (referendum, simple election, multiple choice election, STV election), the text m can be constructed in different ways. We recall from section 2.1.6 that m can be maximum 30 characters in length.

For convenience, we will describe the case of a simple election where the voters have to select one candidate only. Each candidate must have a distinct name which will be used as m and it can be at most 30 characters in length. All names are turned into elliptic curve points using the **String2Point** algorithm. When decoded back from an elliptic curve point, the candidate name will be visible in plain text.

3.3 Election Phase

3.3.1 Voting Procedure

The election phase lasts from the election start date until the election end date. During this time, each voter $\mathcal{V}_j \in \boldsymbol{\mathcal{V}}$ can authenticate using the credentials received over different communication channels from different printing authorities and can participate in the voting process which consists of multiple steps:

- Voter $\mathcal{V}_j$ inputs in the browser all credentials received from the printing authorities $x_{i,j}$, with $i \in \{1, ..., n_\text{p}\}$. The browser combines all credentials to form the voter's secret signing key $x_j = \sum_{i=1}^{n_\text{p}} x_{i,j} \pmod q$.

- The browser tries to authenticate the voter $\mathcal{V}_j$ to the bulletin board $\mathcal{B}$ by following the *voter authentication protocol* (figure 7).

- The voter picks a vote option m and the browser converts it into an elliptic curve point $M \leftarrow \textbf{String2Point}(m)$ (algorithm 1). Next, the browser collaborates with the bulletin board in the *vote cryptogram generation process* in order to encrypt the voter's choice. This process is described in section 3.3.2. At the end of the process, the vote cryptogram e and the proof of correct encryption PK will be generated.

- At this point, the voter has two options. If he has enough trust in the voting application, the voter can register the vote cryptogram e on the bulletin board (the process continues on the next bullet point). Otherwise, the voter has the option to challenge the vote cryptogram (verify that it actually contains the vote m), process described in section 3.3.3.

  If the voter chooses to challenge the encryption, the browser will print on the screen all information about the vote cryptogram that is necessary to revert the encryption process (i.e. the vote cryptogram e, the encryption key Y_enc and the randomizer r). The voter must use a second, **trusted** device to perform the decryption process, in order to output the plain text m. If the vote m corresponds to the selected candidate, the voter gains confidence that his browser behaved correctly, otherwise, there is a clear evidence of an attack to the voter's machine.

  The voter has to recast his vote (return to previous bullet point) and he can repeat this process as many times as needed until he gains enough trust in the voting application.

- The browser asks the bulletin board $\mathcal{B}$ for the latest hash value of the board h_a, which we call *the acknowledged hash*. The browser computes the vote's content hash $h_\text{v} = \mathcal{H}(s)$, where s is a message containing the following information: the voter id, the election id, the vote cryptogram e, the acknowledged hash h_a and the acknowledged time stamp.

- The browser certifies the authenticity of the vote cryptogram by generating a signature of the voter $\mathcal{V}_j$ on the vote's content hash as a *Schnorr*

signature $\sigma \leftarrow \mathbf{Sign}_{x_j}(h_\mathrm{v})$ (algorithm 16). The browser submits to the bulletin board the following: the vote cryptogram e, the proof of correct encryption PK, the vote content hash h_v and the signature σ.

- The bulletin board $\mathcal{B}$ receives this information and accepts the new vote cryptogram if all the following are valid: the proof of correct encryption, the vote content hash, the voter's signature and the acknowledged time stamp is not too old (described in section 3.3.5).

- If the encrypted vote is accepted, the bulletin board $\mathcal{B}$ appends it at the end of the list of vote cryptograms. Next, the bulletin board calculates the new hash value of the list (see section 3.3.5) $h_\mathrm{b} = \mathcal{H}(b)$, where b is a message that contains the following information: the vote content hash h_v, the previous hash value of the board $h_{\mathrm{b}-1}$ and the registration time stamp t_r. Afterwards, the bulletin board sends back to the voter a confirmation receipt ρ in form of a *Schnorr signature* on the following message r: voter's signature σ and the new hash value of the board h_b. The confirmation receipt is computed $\rho \leftarrow \mathbf{Sign}_{x_\mathrm{sign}}(r)$ (algorithm 16).

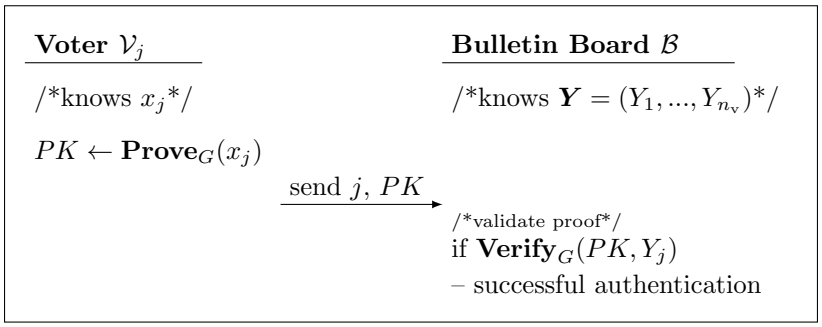

Figure 7: Protocol for voter authentication

3.3.2 Vote cryptogram generation process

During the vote cryptogram generation process, the voter's browser collaborates with the bulletin board for generating the cryptogram. This process results in the fact that the voter will not be in possession of the randomizer value used in the cryptogram e. That is achieved by both the browser and the bulletin board building up the randomizer but none of them knowing its entire value. It is important for the voter not to know this value so he cannot produce cryptographic evidence of the way he voted (as in section 2.4.1), thus enforcing *receipt freeness*. The vote cryptogram generation process is described in figure 8.

The process starts with the Bulletin Board $\mathcal{B}$ delivering to the voter $\mathcal{V}_j$ an empty cryptogram (an encryption of the neutral point $\mathcal{O}$) $e_0 \leftarrow \mathbf{Enc}(\mathcal{O}, r_0)$ (algorithm 10), where $r_0 \in_{\mathrm{R}} \mathbb{Z}_q$. Next, $\mathcal{B}$ starts an *interactive zkp of discrete logarithm equality*, as in figure 2, in order to prove to $\mathcal{V}_j$ that e_0 is indeed an empty cryptogram. Formally,

$$PK_0 = PK[(r_0) : R_0 = [r_0]G \wedge C_0 = [r_0]Y].$$

The reason PK_0 needs to be an *interactive proof* is that PK_0 does not need to be universally valid. Instead, only the voter needs to be convinced that e_0 is an empty cryptogram.

Next, the voter builds his vote cryptogram on top of the empty cryptogram that he just received. Formally, the voter encrypts his vote M with the randomizer $r_1 \in_{\mathrm{R}} \mathbb{Z}_q$ by generating the cryptogram $e_1 \leftarrow \mathbf{Enc}_{Y_{\mathrm{enc}}}(M, r_1)$ (algorithm 10), which he needs to homomorphically add to the empty cryptogram in order to generate his final vote cryptogram $e = \mathbf{AddEnc}(e_0, e_1)$ (algorithm 14). The voter also generates a proof of correct encryption $PK \leftarrow \mathbf{Prove}_G(r_1)$ (algorithm 3).

Note that now, the cryptogram e is actually $\mathbf{Enc}_{Y_{\mathrm{enc}}}(M, r_0 + r_1)$. Both the voter and the bulletin board know part of the randomizer value $r_0 + r_1$ (the voter knows r_1 and the bulletin board knows r_0) but neither of them knows the full value.

When submitting his vote, the voter sends both the cryptogram $e = (R, C)$ and the proof of correct encryption PK. The bulletin board accepts the vote cryptogram if the proof validates $\mathbf{Verify}_G(PK, R - R_0)$ (algorithm 4). This proves that the voter did use the empty cryptogram e_0 in the construction of the vote cryptogram e, thus ensuring that the voter does not know the randomness value of his cryptogram.

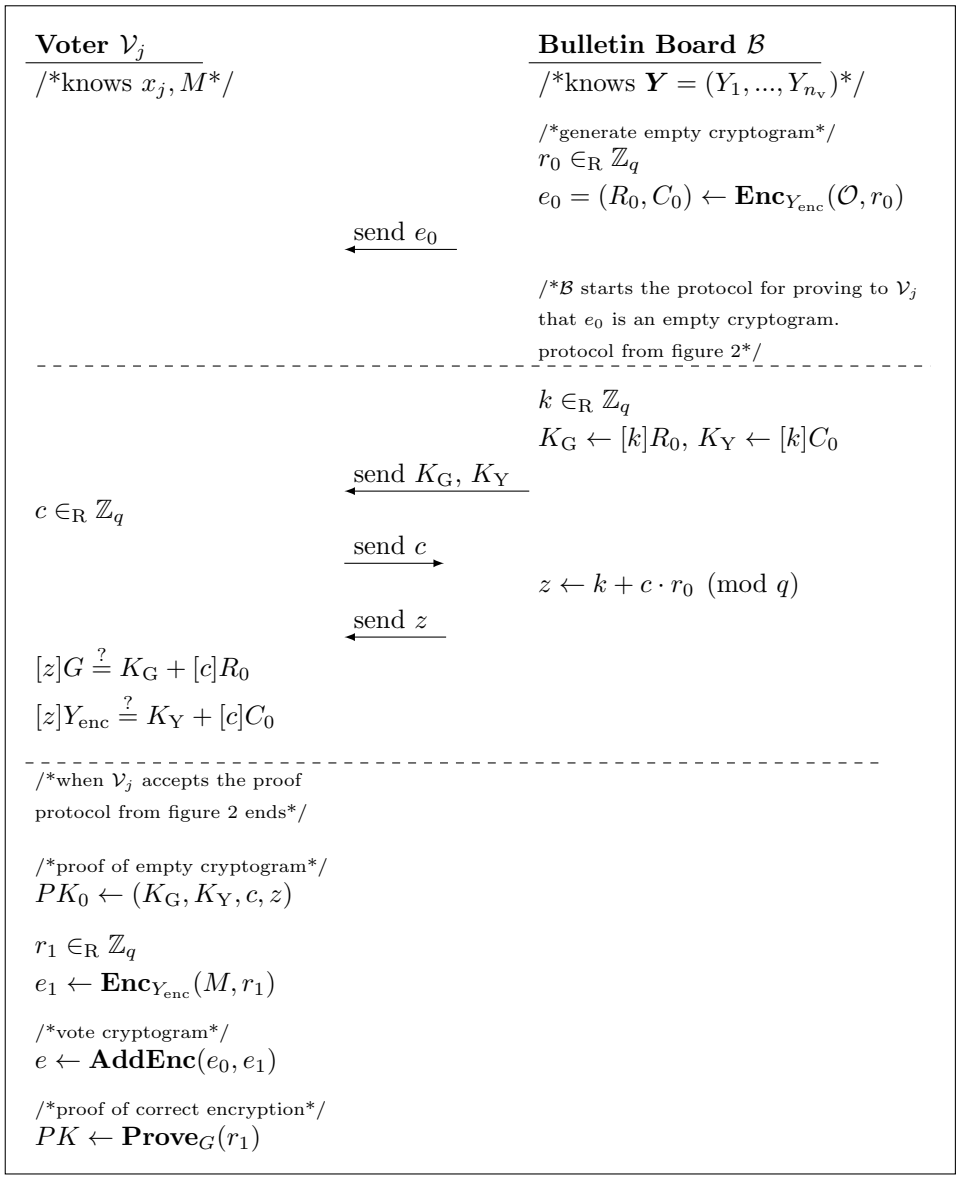

Figure 8: Protocol for generating a vote cryptogram

3.3.3 Challenging a vote cryptogram

By challenging a vote cryptogram we are checking whether the *voting application* is honest or not. In other words, we check whether the voter's browser is under control of an attacker who tries to cast a different vote than the voter's option.

In order to challenge a vote cryptogram e, the browser asks the bulletin board $\mathcal{B}$ for the randomizer value r_0 used to generate the empty cryptogram e_0. Recall from previous section that the browser knows the value r_1 used in the generation of e.

Next, following the *Benaloh challenge* paradigm [12], it will print on the screen (as a QR code or readable format) the vote cryptogram $e = (R, C)$, the encryption key Y_{enc} and the entire randomizer value $r = r_0 + r_1$. All this information must be transferred to another device that the voter **trusts** and that is able to perform the decryption algorithm. Next, the voter needs to verify that the information on the device is identical to the one from the voting application.

If the information matches, the **trusted** device can perform the decryption by extracting the vote $M \leftarrow \mathbf{Dec}_r(e')$ (algorithm 11), where $e' = (Y_{\text{enc}}, C)$. Note that we apply the decryption algorithm on a modified cryptogram where value R and Y_{enc} are swapped. That is done because we are trying to decrypt using the randomizer instead of the decryption key.

Next, the vote is interpreted as plain text $m \leftarrow \mathbf{Point2String}(M)$. If m corresponds to the voter's choice, this proves that the voting application has behaved correctly in the vote encryption process. If there are any differences in the process described above, it means that the voting application has tried to cheat and is now caught.

We argue that, in case the voting application is corrupt and tries to cheat, the procedure of *challenging a vote cryptogram* will discover the cheating attempt with overwhelming probability.

Let's say that the voting application uses the fake option m' instead of m in the process of generating the vote cryptogram. That generates $e' = (R, C) \leftarrow \mathbf{AddEnc}(e_0, e_1)$, where $e_1 \leftarrow \mathbf{Enc}_{Y_{\text{enc}}}(M', r_1)$ and $M' \leftarrow \mathbf{String2Point}(m')$. Now, to cheat the **trusted** device that e' is an encryption of m, it needs to print on the screen as the value of the randomizer $r' \leftarrow r_0 + r_1 - \Delta_{\text{m}}/x_{\text{enc}}$, where $[\Delta_{\text{m}}]G = M - M'$ and x_{enc} is the decryption key.

In conclusion, for the voting application to cheat the **trusted** device, it needs not only the secret decryption key x_{enc} but also, to break the Elliptic Curve Discreet Logarithm Problem in order to find Δ_{m}, which we consider *infeasible*.

3.3.4 Lying about your vote

This section describes why a voter is not able to prove how she voted to a third party verifier, after she has finished the voting process.

Note that, the vote cryptogram e is actually equal to $\mathbf{Enc}_{Y_{\text{enc}}}(M, r_0 + r_1)$ out of which the voter does not know the value r_0. Therefore, the voter would only be able to prove the way she voted based on the empty cryptogram e_0 received from the server, but she cannot prove that e_0 is indeed an empty cryptogram.

Instead, the voter is able to lie about her vote and generate fake evidence to support her claim. Based on the initial proof of empty cryptogram PK_0, the voter can lie about the cryptogram $e = (R, C) \leftarrow \mathbf{Enc}_{Y_{\text{enc}}}(M, r_0 + r_1)$ that it is an encryption of message $M' \in \boldsymbol{M}$, with $M' \neq M$ by computing a fake proof that will support her claim $PK'_0 \leftarrow \mathbf{VoteLie}(PK_0, M, M')$ (algorithm 25). Note that the voter has received both e_0 and PK_0 in the protocol from figure 8.

Algorithm 25: VoteLie(PK_0, M, M')

Data: Proof of empty crypto $PK_0 = (K_{\text{G}}, K_{\text{Y}}, c, z) \in E(\mathbb{F}_p)^2 \times \mathbb{Z}_q \times \mathbb{Z}_q$
The original vote $M \in E(\mathbb{F}_p)$
The fake vote $M' \in E(\mathbb{F}_p)$

`/* compute fake commitment */`

$K'_{\text{Y}} \leftarrow K_{\text{Y}} + [c](M' - M)$

$PK'_0 \leftarrow (K_{\text{G}}, K'_{\text{Y}}, c, z)$

return PK'_0 // $PK'_0 \in E(\mathbb{F}_p)^2 \times \mathbb{Z}_q \times \mathbb{Z}_q$

The voter delivers the vote cryptogram $e = (R, C)$, the fake empty cryptogram $e'_0 = (R_0, C'_0)$, where $C'_0 = C_0 + M - M'$, the fake proof of empty cryptogram PK'_0 and a proof of correct re-encryption $PK \leftarrow \mathbf{ProveEquality}_{G,Y_{\text{enc}}}(r_1)$ (algorithm 5). To be convinced that the cryptogram e contains vote M' a verifier checks the proof of re-encryption $\mathbf{VerifyEquality}_{G,Y_{\text{enc}}}(PK, R - R_0, C - C'_0 - M')$. If it validates, the verifier checks the second proof of empty cryptogram $\mathbf{VerifyEquality}_{G,Y_{\text{enc}}}$ (PK'_0, R_0, C'_0) (algorithm 6). If both proofs validate, verifier is convinced about the fake evidence.

Therefore, because the voter is able to produce valid fake evidence about the way she voted, a voter should not be trusted for proving the way she voted regardless of the fact that she is honest or not.

3.3.5 Append-only Bulletin Board

All vote cryptograms received by the Bulletin Board are published on the list of vote cryptograms together with a signature of the voter and the *board hash value*. They are stored as an *append only list*, that means, no vote cryptograms are removed or replaced and each new vote cryptogram is appended at the end of the list. The structure of the Bulletin Board has been inspired from [13].

The process of submitting a vote to the Bulletin Board starts by the voter asking for the latest hash value of the board. The Bulletin Board returns the current hash value of the board $h_{\mathrm{a},i}$ and the current time stamp $t_{\mathrm{a},i}$, which will be used as parameters in the generation of the vote submission. The entire process can be seen in figure 9.

Each time a new submission of a vote cryptogram $(e_i, h_{\mathrm{v},i}, \sigma_i, PK_i)$ is received, the Bulletin Board validates the following:

- the vote submission is not too old, i.e. $t_{\mathrm{a},i} < t_{\mathrm{r},i} < t_{\mathrm{a},i} + \epsilon$, where $t_{\mathrm{r},i}$ is the registration timestamp and ϵ is the *latency parameter*, which represents the maximum time the vote submission process can take,
- the authenticity of the vote submission, i.e. σ_i is a valid, well-formatted signature,
- the correctness of the vote cryptogram, i.e. validate that e_i is constructed based on the empty cryptogram $e_{0,i}$ by checking the proof of correct encryption PK_i.

If all validations succeed, the vote submission is registered (as the i^{th} item on the list) and a new board hash value is calculated $h_{\mathrm{b},i} = \mathcal{H}(h_{\mathrm{v},i}||h_{\mathrm{b},i-1}||t_{\mathrm{r},i})$, where e_i is the vote cryptogram, $h_{\mathrm{v},i}$ is the vote's content hash, σ_i is the voter's signature on the vote submission and $t_{\mathrm{r},i}$ is the registration time stamp.

It is visible that the board hash value is calculated based on the previous board hash value in a *blockchain* like manner where each vote received is a block in the chain. The first element of the bulletin board will compute its board hash value by using as the previous board hash the value $h_{\mathrm{b},0} = 0$ (*genesis hash*).

The difference between the protocol described in paper [13] and our protocol is that we accept multiple vote submissions coming at the same time, i.e. vote submissions that have the same value for the acknowledged hash h_{a}. This is an unusual situation that can happen when two voters "race" to submit their votes and one of them has a slow internet connection while the other has a faster one. The bulletin board accepts the second (losing) vote submission, even if its acknowledged hash h_{a} does not point to the previous item from the bulletin board. The list aspect of the board is maintained by computing the *hash value of the board* each time a new vote submission is registered.

Each bulletin board item contains the following information: the voter's id, the vote cryptogram, the acknowledged board hash and the acknowledged timestamp, the voter's signature, the registration timestamp and the new board hash (the hash value of the board after this item has been appended).

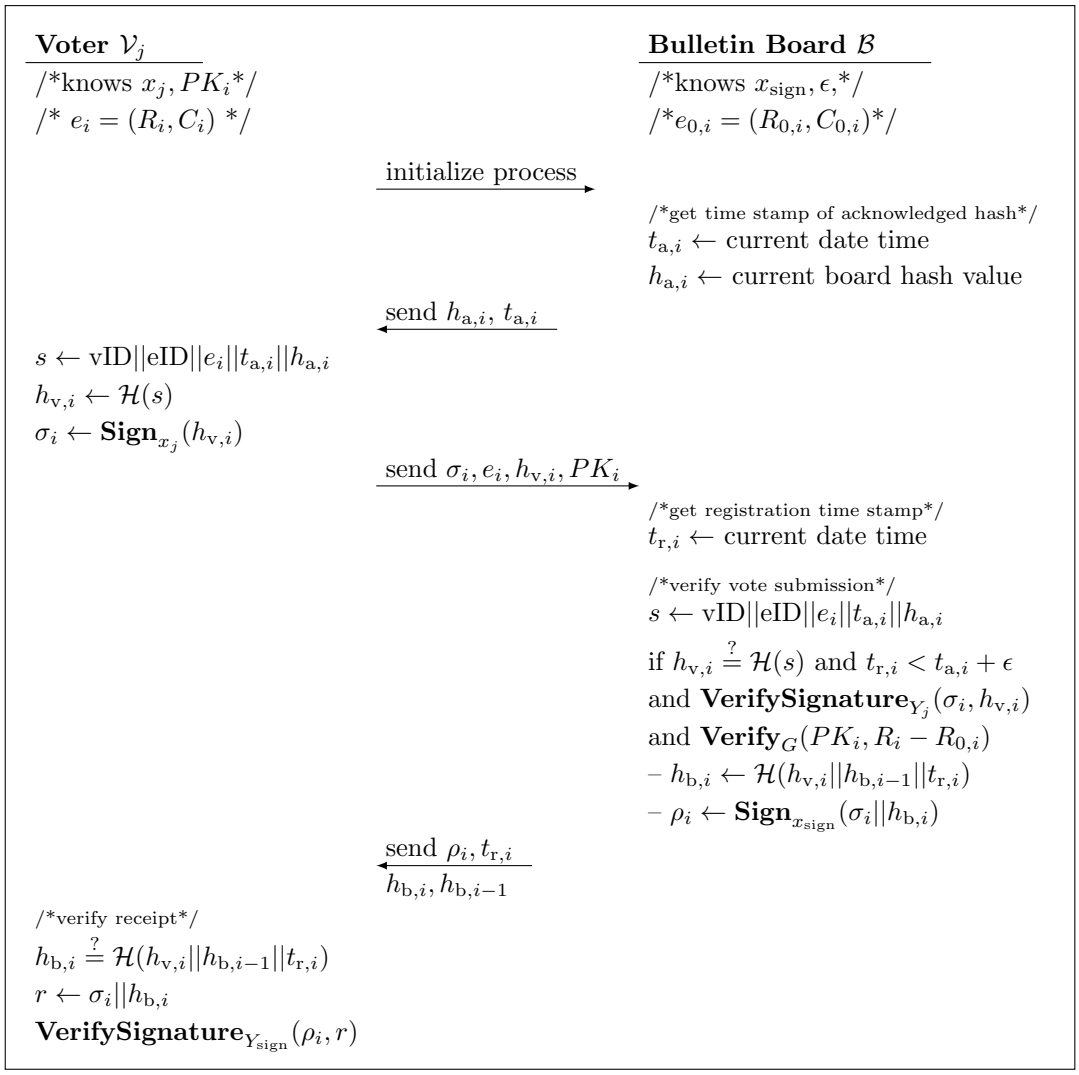

Figure 9: Protocol for submitting vote cryptograms

3.3.6 System events

Election configuration belongs also to the bulletin board. An election administrator is able to generate a special kind of bulletin board item, called *a system event*, that describes the election context. For example, such system events can define the following:

- *election configuration* that includes: the election signature verification key Y_{sign}, the encryption key Y_{enc}, the election start and closing dates,

- *ballot configuration* that includes: the question of the election and all candidate names $(m1, ..., m_{n_c})$, where n_c is the total number of candidates,
- *voters configuration* that includes: the list of eligible voters, each represented by an id and a signature verification key Y_i, where $i \in \{1, ..., n_v\}$ and n_v is the total number of voters,
- *threshold configuration* that includes: the threshold limit t and the list of all trustees, each represented by an id, a public key Y_i and the list of public polynomial coefficients $P_{i,k}$, where $i \in \{1, ...n_t\}$, $k \in \{1, ..., t-1\}$ and n_t is the total number of trustees (recall from section 2.4.3).

The structure of a system event follows the structure of a vote submission. A system event has a registration time $t_{r,i}$, a content hash $h_{c,i}$ and a signature $\sigma_i \leftarrow \mathbf{Sign}_{x_{sign}}(h_{c,i})$ (algorithm 16), which certifies that the event has been generated by the bulletin board. Recall from section 3.2 that x_{sign} is the bulletin board's signing key. When the event has been appended on the bulletin board, a new board hash value is computed, by following the same strategy: $h_{b,i} = \mathcal{H}(h_{c,i}||h_{b,i-1}||t_{r,i})$, where $h_{b,i-1}$ is the previous board hash value.

The election process starts by having the election context locked down on the bulletin board in form of system events: an *election configuration*, a *ballot configuration*, a *voters configuration* and a *threshold configuration* events. If any change happens to the election context during the election process, a system event that describes the change is generated and appended on the board. That change will be effective in the election process from that moment on.

3.3.7 Vote Confirmation Receipt

Once the vote cryptogram e_i has been registered, the voter receives back from the bulletin board a confirmation receipt in form of a Schnorr signature (algorithm 16) $\rho_i = \mathbf{Sign}_{x_{sign}}(\sigma_i||h_{b,i})$, together with the new board hash value $h_{b,i}$, the previous board hash value $h_{b,i-1}$ and the registration time stamp $t_{r,i}$. Recall from section 3.2 that x_{sign} is the bulletin board's signing key.

The receipt certifies that the vote cryptogram has been registered on the bulletin board at exactly version $h_{b,i}$. The voter can use this receipt to verify, at any time, that her vote is included on the board, and that the history of the board has not changed by validating the hash value.

If the previous board hash value $h_{b,i-1}$ does not match with the acknowledged hash $h_{a,i}$, that means a "race" situation (described in the previous section) has happened and the current voter has lost the race, i.e. while the current voter was preparing her vote submission, another voter has successfully managed to post another vote submission. This is a perfectly valid scenario that can occur in busy situations.

Note that, if a voter has a confirmation receipt that does not correspond with the current state of the bulletin board, that immediately reveals an attempt to break the integrity of the bulletin board and should be reported to the election authorities.

3.4 Post-election Phase

After the voting time has finished, the election proceeds to the last phase which will generate the result of the election. Now, the bulletin board $\mathcal{B}$ does not accept any new vote cryptograms anymore. The list of votes remains publicly available for voters to check that their vote cryptogram is included (using their confirmation receipt) and for auditors to check that the hash values of the list are consistent (the integrity of the board is persistent).

In this chapter we will present all the steps that take place in the post-election phase.

3.4.1 Cleansing Procedure

The bulletin board $\mathcal{B}$ will create a different list that contains only the cryptograms of the *valid cryptograms* from the bulletin board. A *valid cryptogram* is a vote cryptogram item on the bulletin board that has a valid signature and that has not been overwritten by a latter vote cryptogram generated by the same voter.

The new list of cryptograms is called *the initial mixed board* and is saved in a file. This file will be used as the input to the *mixing phase*.

The cleansing procedure is publicly auditable as both the list of vote cryptograms and the initial mixed board are publicly available.

3.4.2 Mixing Phase

During the mixing phase, the list of cryptograms will change its appearance several times, being shuffled in an indistinguishable way. Each mixer from the mixnet $\mathcal{M}_i \in \boldsymbol{\mathcal{M}}$ applies its mixing algorithm in sequential order (the output of $\mathcal{M}_{i-1}$ will be used as input for $\mathcal{M}_i$). The first mixer $\mathcal{M}_1$ applies its algorithm on *the initial mixed board* and the output of the last mixer $\mathcal{M}_{n_\mathrm{m}}$ is used as *the final mixed board*.

Obviously, each mixer $\mathcal{M}_i$ knows the shuffling coefficients of its own mixing algorithm and it is able to link the votes on the mixed board at step $i-1$ (input board) with the ones on the mixing board at step i (output board). However, $\mathcal{M}_i$ does not know the shuffling coefficients of the other mixers algorithms so it cannot create a full link between the votes on the final mixed board and the ones on the initial mixed board, unless all mixers are corrupt and collude against the election. It is important this full link to be unknown because the votes on the initial mixed board can be associated with an identity (through its digital signature).

Thus, on the assumption that at least one mixer is honest, during the mixing phase the link between a vote on the final mixed board and its owner (voter) is broken. Therefore, mixing phase preserves anonymity.

Each mixer $\mathcal{M}_i \in \boldsymbol{\mathcal{M}}$ asks the bulletin board $\mathcal{B}$ for the latest version of the mixed board. The bulletin board $\mathcal{B}$ will grant permission to one mixer at a time. When $\mathcal{M}_i$ is granted permission, it receives the mixed board $\boldsymbol{e_{i-1}}$. Next, $\mathcal{M}_i$ follows the mixing procedure to compute the new mixed board by randomly

permuting the entire list of cryptograms and re-encrypting each cryptogram from the permuted list with distinct randomnizers. Finally, $\mathcal{M}_i$ computes a proof of correct mixing in form of Furukawa's Proof of Shuffle $(\boldsymbol{e_i}, PK) \leftarrow \textbf{MixingProcedure}_{Y_{\text{enc}}}(\boldsymbol{e_{i-1}})$ described in algorithm 26.

We recall from section 2.6 that the algorithm used for proving a shuffle has the soundness property on the assumption that the prover is not in the possession of the randomizers used in the generation of each cryptogram. We consider this assumption sane as the cryptograms are generated in collaboration between the bulletin board and each voter (figure 8). By following this protocol, we ensure that there is no single entity that knows the randomizer value behind any cryptogram.

Algorithm 26: MixingProcedure$_{Y_{\text{enc}}}(\boldsymbol{e_{i-1}})$

Data: The input mixed board $\boldsymbol{e_{i-1}} \in \mathbb{E}^n$
$\boldsymbol{r} \in_{\text{R}} \mathbb{Z}_q^n$
$\psi \in_{\text{R}} \Psi_n$
`/* compute new mixed board */`
$\boldsymbol{e_i} \leftarrow \textbf{Shuffle}_{Y_{\text{enc}}}(\boldsymbol{e_{i-1}}, \boldsymbol{r}, \psi)$ `// algorithm 18`
`/* compute shuffle proof */`
$PK \leftarrow \textbf{ProveShuffle}_{Y_{\text{enc}}}(\boldsymbol{e_{i-1}}, \boldsymbol{r}, \psi)$ `// algorithm 23`
return $(\boldsymbol{e_i}, PK)$ // $(\boldsymbol{e_i}, PK) \in \mathbb{E}^n \times (E(\mathbb{F}_p)^{8+4\cdot n} \times \mathbb{Z}_q^n \times \mathbb{Z}_q^{2+n})$

Note that, while one mixer applies its mixing algorithm on a mixing board, the other mixers have to wait until the new mixing board has been approved.

Once the mixing procedure has finished, both the mixed board $\boldsymbol{e_i}$ and the mixing proof PK are saved in separate files. The mixer uploads both files to the bulletin board which checks the validity of the proof by running the algorithm $\textbf{VerifyShuffle}_{Y_{\text{enc}}}(PK, \boldsymbol{e_{i-1}}, \boldsymbol{e_i})$ (algorithm 24). If the proof validates, the bulletin board $\mathcal{B}$ accepts $\boldsymbol{e_i}$ as the new version of the mixed board.

When all mixers have applied their mixing procedures, the final mixed board is used in the decryption phase to compute the election results.

3.4.3 Decryption Phase

Because the link between a vote cryptogram and its voter has been broken during the mixing phase, it is safe now to decrypt all the cryptograms from the final mixed board as it does not violate the secrecy of the election. Furthermore, decrypting the final mixed board would lead to accurate and correct results as it contains the exact same votes as the initial bulletin board, fact proven by the mixing proofs.

During the decryption phase, trustees have to collaborate again. This time, only a threshold of trustees are sufficient to decrypt the votes. The threshold value t is the one set during the *threshold ceremony*. Recall that each trustee $\mathcal{T}_i \in \boldsymbol{\mathcal{T}}$ is in possession of a share of the election decryption key sx_i.

Assuming that the subset of trustees $\boldsymbol{T} \subset \boldsymbol{\mathcal{T}}$ participate in the decryption phase, with the cardinality of $\boldsymbol{T}$ being $n_{\mathrm{d}} \geq t$. Each trustee $\mathcal{T}_i \in \boldsymbol{T}$ downloads from the bulletin board $\mathcal{B}$ the final mixed board $\boldsymbol{e} = (e_1, ..., e_{n_{\mathrm{e}}})$ and computes partial decryptions of every single cryptogram by following the algorithm $\boldsymbol{S_i} = (S_{i,1}, ..., S_{i,n_{\mathrm{e}}}) \leftarrow \textbf{PartialDecryption}_{sx_i}(\boldsymbol{e})$ (algorithm 27).

Algorithm 27: PartialDecryption$_{sx_i}(\boldsymbol{e})$

Data: The share of decryption key $sx_i \in \mathbb{Z}_q$
The board of cryptograms $\boldsymbol{e} = (e_1, ..., e_{n_{\mathrm{e}}}) \in \mathbb{E}^{n_{\mathrm{e}}}$)
$e_j = (R_j, C_j), j \in \{1, ..., n_{\mathrm{e}}\}$
for $j \leftarrow 1$ **to** n_{e} **by** 1 **do**
  $S_{i,j} \leftarrow [sx_i]R_j$
end
$\boldsymbol{S_i} \leftarrow (S_{i,1}, ..., S_{i,n_{\mathrm{e}}})$
return $\boldsymbol{S_i}$ // $\boldsymbol{S_i} \in E(\mathbb{F}_p)^{n_{\mathrm{e}}}$

After computing partial decryptions, trustee $\mathcal{T}_i$ generates a proof of correct decryption in form of a *non-interactive Proof for Multiple Discrete Logarithms* (section 2.2.3) $PK_i \leftarrow \textbf{ProveMultiple}_{G,R_1,...,R_{n_{\mathrm{e}}}}(sx_i)$ (algorithm 7). The trustee sends to the bulletin board both the partial decryptions $\boldsymbol{S_i}$ together with the proof of correct decryption PK_i. The bulletin board $\mathcal{B}$ accepts the partial decryption $\boldsymbol{S_i}$ if the proof of correct decryption validates according to the algorithm $\textbf{VerifyMultiple}_{G,R_1,...,R_{n_{\mathrm{e}}}}(PK_i, sY_i, S_{i,1}, ..., S_{i,n_{\mathrm{e}}})$ (algorithm 8). Recall from section 2.4.3 that trustee's public share of the decryption key sY_i is publicly available.

Upon receiving all partial decryptions $(\boldsymbol{S_1}, ..., \boldsymbol{S_{n_{\mathrm{d}}}})$ from all trustees in $\boldsymbol{T}$, the bulletin board $\mathcal{B}$ aggregates all partial decryptions for each cryptogram $e_j \in \boldsymbol{e}$ to finalize the decryption and extract the votes $\boldsymbol{V} = (V_1, ..., V_{n_{\mathrm{e}}})$. The procedure for aggregating all partial decryptions is described in the algorithm $\boldsymbol{V} \leftarrow \textbf{FinalizeDecryption}(\boldsymbol{e}, \boldsymbol{S_1}, ..., \boldsymbol{S_{n_{\mathrm{d}}}})$ (algorithm 28).

Algorithm 28: FinalizeDecryption$(\boldsymbol{e}, \boldsymbol{S_1}, ..., \boldsymbol{S_{n_{\mathrm{d}}}})$

Data: The board of cryptograms $\boldsymbol{e} = (e_1, ..., e_{n_{\mathrm{e}}}) \in \mathbb{E}^{n_{\mathrm{e}}}$
$e_j = (R_j, C_j), j \in \{1, ..., n_{\mathrm{e}}\}$
The partial decryptions $\boldsymbol{S_i} = (S_{i,1}, ..., S_{i,n_{\mathrm{e}}}) \in E(\mathbb{F}_p)^{n_e}, i \in \boldsymbol{T}$
for $j \leftarrow 1$ **to** n_{e} **by** 1 **do**
  $V_j \leftarrow C_j - \sum_{i \in \boldsymbol{T}} [\lambda(i)] S_{i,j}$ // $\lambda(i)$ computed as in figure 5
end
$\boldsymbol{V} \leftarrow (V_1, ..., V_{n_{\mathrm{e}}})$
return $\boldsymbol{V}$ // $\boldsymbol{V} \in E(\mathbb{F}_p)^{n_{\mathrm{e}}}$

At the end of the decryption phase we will be in possession of the *raw result* of the election, i.e. the full list of votes in plain text (decrypted form).

3.4.4 Result Publication

The *results module* is responsible for interpreting the *raw result* and present the result of the election in a more readable way. The interpretation of the result is dependant on the election type (simple election, multiple election, STV, etc.).

For simplicity, we will consider the simple election case, where voters had to choose one option from a predefined set of candidates.

The bulletin board $\mathcal{B}$ verifies every single vote from the list of raw results $V_i \in \boldsymbol{V}$ and if it matches one of the vote options $V_i \in \boldsymbol{M}$ the vote counter for that particular candidate is incremented. If a vote from the list of raw results does not correspond to any of the vote options $V_i \notin \boldsymbol{M}$, that represents an invalid vote and the counter of invalid votes is incremented.

Results will be published as a list of candidates next to the amount of votes they received, in decending order of the vote counter.

4 Auditing process

This section describes the entire auditing process of an election, i.e. all the verification mechanisms, who conducts them and what cryptographic algorithms they are made of. These verification mechanisms can be split into two categories:

- individually verifiable
- publicly verifiable

Individual verification mechanisms are targeted to one single person and allows him to verify one single piece of information. This type of verification mechanism is used only when the piece of information in question is relevant only to that single individual, for example, a voter verifies that his voting application behaves correctly.

Public verification mechanisms are accessible to anybody. They are used to validate that the entire election process behaves correctly. This kind of mechanisms is typically run by certified auditors that will validate or invalidate an election result. Nevertheless, they could be run by any public person that has access to the right verification algorithms.

The system provides verification mechanisms of the following aspects:

- vote is cast as intended
- vote is registered as cast
- votes are counted as registered
  - mixing procedure
  - decryption
- eligibility of the registered votes
- integrity of the bulletin board

4.1 Individual verification mechanisms

During the voting process, the voter is able to verify two aspects of his own vote: it is cast as intended and it is registered as cast.

4.1.1 Vote is cast as intended

In order to verify that the vote is cast as intended, the voter needs to verify that the encryption mechanism of the voting application behaved correctly, i.e. the cryptogram contains the correct vote. After generating the vote cryptogram, the voter can choose whether to submit his vote or to challenge the encryption process. If the voter chooses to challenge the vote cryptogram, the system will print on the screen the value of the cryptogram $e = (R, C)$, the encryption key Y_{enc}, and the randomizer r used in the encryption. Note that $r = r_0 + r_1$,

where r_0 is the randomizer generated by the server and r_1 is generated by the voting application. That means the server has to collaborate in this process for providing r_0.

Now the voter can use a secondary device to decrypt the content of the cryptogram by applying $M \leftarrow \mathbf{Dec}_r(e')$ (algorithm 11), where $e' = (Y_{\text{enc}}, C)$. If the vote M corresponds to the correct value that the voter intended to cast, then the voter gains confidence that the voting application behaves correctly.

If the voter chooses to challenge the vote cryptogram, then the cryptogram is invalidated because the value r_0 has been exposed. After challenging the vote cryptogram, the voter has to recast his vote by generating another vote cryptogram, which again, he has the option to challenge or submit.

4.1.2 Vote is registered as cast

After posting a vote submission (vote cryptogram e and the voter signature σ), the voter receives a receipt ρ that certifies that his vote submission has been registered on the bulletin board at position h_{b}. The receipt can be validated by checking $\mathbf{VerifySignature}_{Y_{\text{sig}}}(\rho, \sigma || h_{\text{b}})$ (algorithm 17).

Anytime during the election, the voter can check his receipt against the bulletin board, which responds with the appropriate vote submission, thus the voter gains confidence that his vote is registered as cast.

4.2 Public auditing process

During the voting process, any public auditor is able to verify the eligibility of the registered votes (i.e. all vote submissions were generated by a genuine voter) and the integrity of the bulletin board (i.e. no registered vote submissions have been removed or tampered with).

After the voting phase has finished and the integrity of the election has been confirmed, the bulletin board goes through the cleansing procedure, where some of the vote submissions are filtered out based on some public rules. For each of the remaining vote submissions, the system will consider only its vote cryptogram which will be passed to the following counting process. The rules of the cleansing procedure are the following:

- only vote submissions with valid digital signatures are considered
- for each voter, only his last vote submission is considered
- if a voter has been removed, all his vote submissions are disregarded

During the counting process, any public auditor is able to verify that the result is counted based on the registered votes. This verification process is made out of two parts: verification of the mixing procedure (i.e. confirm that vote cryptograms have not been tampered with during the mixing phase) and verification of the decryption process (i.e. confirm that all trustees correctly decrypted the vote cryptograms).

All public auditing processes are based on the data provided by the public bulletin board, which includes:

- all registered vote submissions, each represented by the following data: the voter id i, the vote cryptogram e, the acknowledged hash h_{a}, the acknowledged time stamp t_{a}, the voter digital signature σ, the board hash h_{b} and the registration time stamp t_{r}
- all the election configuration, including the election id eID, the election encryption key Y_{enc} and signature verification key Y_{sig} and the signature verification keys of all eligible voters Y_i, with $i \in \{1, ..., n_{\mathrm{v}}\}$, where n_{v} is the total number of voters
- all the data regarding the threshold ceremony, including the threshold limit t, the public keys of all trustees Y_j and the public threshold coefficients $P_{j,k}$, where $j \in \{1, ..., n_{\mathrm{t}}\}$ and $k \in \{1, ..., t-1\}$

4.2.1 Eligibility verifiability

Any public auditor is able to validate the digital signature of each vote submission. A valid digital signature confirms that the vote submission has been genuinely generated by an eligible voter. The signature validation algorithm is $\mathbf{VerifySignature}_{Y_i}(\sigma, h_{\mathrm{v}})$ (algorithm 17), where $h_{\mathrm{v}} = \mathcal{H}(i||eID||e||t_{\mathrm{a}}||h_{\mathrm{a}})$.

Note that through his digital signature, the voter also certifies the history of the bulletin board (i.e. at time t_{a} the bulletin board had its hash value h_{a}).

4.2.2 Integrity of the bulletin board

Any public auditor is able to validate the integrity of the bulletin board by checking that the board hash of each item on the board is computed based on the previous board hash, in a blockchain manner. The calculation below include an extra index j to specify the order of the entries from the bulletin board. To validate the integrity of the board, for each $j \in \{1, ..., n_{\mathrm{b}}\}$, where n_{b} is the total number of items on the bulletin board, the following has to match: $h_{\mathrm{b},j} = \mathcal{H}(h_{\mathrm{v},j}||h_{\mathrm{b},j-1}||t_{\mathrm{r},j})$.

Note that each board hash $h_{\mathrm{b},j}$ is computed based on its previous board hash $h_{\mathrm{b},j-1}$. The value of the initial board hash is $h_{\mathrm{b},0} = 0$.

4.2.3 Verification of the cleansing procedure

Any public auditor is able to verify the cleansing procedure as both the bulletin board file and the cleansed list of vote cryptograms are publicly available. The auditor has to apply the cleansing rules on the bulletin board and verify that the output is identical with the list of vote cryptograms.

4.2.4 Verification of mixing procedure

During and after the mixing phase, any public auditor is able to verify the mixing procedure of each mix node $\mathcal{M}_i$, with $i \in \{1, ..., n_{\text{m}}\}$, where n_{m} is the total number of mix nodes, by applying $\textbf{VerifyShuffle}_{Y_{\text{enc}}}(PK_i, \boldsymbol{e_{i-1}}, \boldsymbol{e_i})$ (algorithm 24), where PK_i is the proof of correct mixing, $\boldsymbol{e_i}$ is the mixed board of cryptograms and $\boldsymbol{e_{i-1}}$ is the previous mixed board of cryptograms.

Note that $\boldsymbol{e_0} = \{e_1, ..., e_{n_{\text{cb}}}\}$ is the initial, **cleansed** board of vote cryptograms as registered by the voters, where n_{cb} is the total number of cleansed cryptograms.

4.2.5 Verification of the decryption

During and after the decryption phase, any public auditor is able to verify each partial decryption $\boldsymbol{S_i} = \{S_{i,1}, ..., S_{i,n_{\text{cb}}}\}$ computed by the trustee $\mathcal{T}_i$, with $i \in \{1, ..., n_{\text{t}}\}$, where n_{t} is the total number of trustees.

Note that each trustee applies his partial decryption on the final mixed board of cryptograms $\boldsymbol{e_{n_m}} = \{e_1, ..., e_{n_{\text{cb}}}\}$ (the board outputted by the last mix node), with each $e_j = (R_j, C_j)$, where $j \in \{1, ..., n_{\text{cb}}\}$.

To validate a partial decryption, the auditor has to run the verification procedure $\textbf{VerifyMultiple}_{G, R_1, ..., R_{n_{\text{cb}}}}(PK_i, sY_i, S_{i,1}, ..., S_{i,n_{\text{cb}}})$ (algorithm 8), where PK_i is the proof of correct decryption and sY_i is the trustee's public share of the decryption key. Note that sY_i is publicly computable as described in section 2.4.3.

To validate the aggregation of all partial decryptions $(\boldsymbol{S_1}, ..., \boldsymbol{S_{n_d}})$, where n_{d} is the total number of trustees that participate in the decryption phase, any public auditor can run the algorithm $\boldsymbol{V} \leftarrow \textbf{FinalizeDecryption}(\boldsymbol{e_{n_m}}, \boldsymbol{S_1}, ..., \boldsymbol{S_{n_d}})$ (algorithm 28) that will output $\boldsymbol{V} = \{V_1, ..., V_{n_{\text{cb}}}\}$, the raw result of the election (i.e. the list of decrypted votes). The auditor has to compare $\boldsymbol{V}$ with the list of votes published by the system, which should be identical.

Counting the votes and sorting the candidates based on their vote count is trivial.

5 Election properties

In this section, we describe what security properties our election system has. We also describe what each of these properties means and under what context the are reached.

5.1 Mobility

The voter can use any device (PC, laptop, tablet, smart phone), that he has and he trusts, to connect to the election system. The voter does not need to be in a special location (e.g. polling station) in order to vote. Instead, the voter can participate in the voting process being located in any place, that he considers secure and private, and that has an internet connection.

We claim *mobility* as a property of Assembly Voting X.

5.2 Eligibility

Eligibility is defined as the fact that only a limited number of predefined voters are allowed to cast a valid vote.

During the *pre-election phase* (section 3.2), the *election administrators* define the list of eligible voters $\boldsymbol{\mathcal{V}} = (\mathcal{V}_1, ..., \mathcal{V}_{n_\text{v}})$, where n_v is the total number of voters, and the list of printing authorities $\boldsymbol{\mathcal{P}} = (\mathcal{P}_1, ..., \mathcal{P}_{n_\text{p}})$, where n_p is the total number of printing authorities.

During the *voter credential distribution process* (section 3.2.1), all *printing authorities* assign a distinct public signature verification key Y_i to each voter $\mathcal{V}_i$. At the same time, each *printing authority* $\mathcal{P}_j \in \boldsymbol{\mathcal{P}}$ distributes to each voter their voting credentials $x_{i,j}$, with $i \in \{1, ..., n_\text{v}\}$ and $j \in \{1, ..., n_\text{p}\}$ such that $[\sum_{j=1}^{n_\text{p}} x_{i,j}]G = Y_i$. After receiving all his election credentials $x_{i,j}$, voter $\mathcal{V}_i$ can compute his private signing key $x_i = \sum_{j=1}^{n_\text{p}} x_{i,j}$.

When submitting a vote, the voter $\mathcal{V}_i$ digitally signs the vote submission with his private signing key x_i and the election system accepts the vote submission only if the digital signature matches the voter's public signature verification key Y_i. By following this process, we make sure that the vote submission was generated by somebody in possession of x_i, which could only be the voter.

One can notice that the printing authority $\mathcal{P}_j$ knows a part of the signing key of voter $\mathcal{V}_i$ but not enough. Therefore, we argue that Assembly Voting X has the *eligibility property* on the assumption that there exist multiple printing authorities that do not communicate with each other during the election process.

5.3 Privacy

By the term *privacy of the election* we understand that nobody is able to read any votes or any part of the result unless it is supposed to, i.e. it is specified in the election protocol.

By following the election protocol, it is guaranteed that:

- the secrecy of the vote is preserved. No votes from the public bulletin board are decrypted, therefore no connection between a vote and a voter can be made.
- no partial results are computed during the election process. A result is calculated only once, after the election phase has finished.

All votes, that are posted on the public bulletin board, are encrypted using the ElGamal cryptosystem based on elliptic curve cryptography (more details in sections 2.1 and 2.4). Moreover, using a *t out of n* threshold encryption scheme (section 2.4.3), we enforce that there is no single entity that can perform the decryption of any data from the public bulletin board, but instead, it is needed a group of minimum t trustees to collaborate.

Therefore, we claim that Assembly Voting X has the *privacy property* on the assumption that at least t trustees are honest, with $t > n/2$ and $n > 2$.

One can argue that, because the bulletin board data is public, somebody could save all the data for long enough until the elliptic curve cryptosystem will be broken, and so will be able to decrypt all the data contrarily to our protocol. This fact demonstrates that our system does not comply to the *everlasting privacy property*. We take note of this fact and we accept it.

5.4 Anonymity

The anonymity property implies the fact that nobody knows the connection between a voter identity and its decrypted vote from the final *raw result* list of votes. This property is reached by implementing a mixnet of mixers that sequentially shuffle the list of vote cryptograms in an indistinguishable way, before they get decrypted (section 3.4.2).

Obviously, each mixer knows the way it shuffled the list of cryptograms but it does not know how it was shuffled by the other mixers. Thus, it is important that mixers do not communicate with each other.

We claim that Assembly Voting X is an *anonymous voting system* on the assumption that there is a mixnet of multiple mixers out of which at least one is honest.

5.5 Integrity

We define the *integrity* of an election as the property that all data recorded during the election is never modified or deleted.

The integrity of the election is preserved in our system by publishing all events (vote submissions or system events) on the bulletin board. Moreover, the bulletin board has a *blockchain-like* structure that guarantees that the history of the bulletin board never changes. Also, the voters act like miners of the blockchain whenever they submit a new vote cryptogram, by signing on the history of the blockchain.

Every time a new vote submission is appended on the bulletin board, the voter receives a vote receipt ρ_i that contains a pointer to the item on the bulletin board, called *the board hash value* $h_{\mathrm{b},i}$. This value is computed based on the previous *board hash value* $h_{\mathrm{b},i-1}$, which is computed based on the one before, and so on, until it reaches the *genesis hash*, which is 0. This means that every time a voter checks his voter receipt, the entire bulletin board history is validated.

We claim that Assembly Voting X achieves the *integrity property* through the public bulletin board construction.

5.6 Verifiability

All steps of the election protocol are verifiable (more details in section 4). There are two levels of verifiability that can be performed by different actors. Some steps are individually verifiable (i.e. only the voter that is currently performing this step can verify that the process is happening correctly), such as:

- verify that the vote is cast as intended
- verify that the vote is registered as cast

The rest of the steps from the election protocol are publicly verifiable:

- the threshold ceremony
- the public bulletin board history
- the integrity and eligibility of each vote submission
- the integrity of each system event
- the correctness of the cleansing procedure, mixing phase and decryption phase (verification that votes are counted as registered)

We claim that Assembly Voting X is a *verifiable election system*.

5.7 Receipt-freeness

We define the *receipt-free* property as the fact that a voter is not able to prove to a third party the way he voted, after he submitted his vote cryptogram.

During the *vote cryptogram generation process* (section 3.3.2), the voter receives from the bulletin board an empty cryptogram e_0 that he uses to generate his final vote cryptogram e in order to encrypt his vote M. At the end of the process, the vote cryptogram e would be equal to $\mathbf{Enc}_{Y_{\text{enc}}}(M, r_0 + r_1)$, where r_0 is known by the bulletin board and r_1 is known by the voter. The voter is convinced that e_0 is an empty cryptogram because of a interactive proof PK_0 generated between the voter and the bulletin board.

The empty cryptogram e_0 and the proof PK_0 are relevant only for the communication amongst the voter and the bulletin board, to make sure that none of them has the entire randomness value $r_0 + r_1$ that is used to generate the cryptogram e. Therefore, e_0 and PK_0 are not included on the public bulletin board, thus not publicly available.

After the vote cryptogram e has been accepted on the board, the voter is not able to produce valid cryptographic evidence that e is an encryption of M referencing only the data that is publicly available. More details are described in section 3.3.4.

Therefore, we claim that Assembly Voting X is a *receipt-free* voting protocol.

References

[1] René Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44(170):483–494, 1985.

[2] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[3] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[4] Sherman S. M. Chow, Changshe Ma, and Jian Weng. Zero-knowledge argument for simultaneous discrete logarithms. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics*, pages 520–529, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 522–526, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[6] Yvo G. Desmedt and Yair Frankel. Threshold cryptosystems. In *Proceedings on Advances in Cryptology*, CRYPTO '89, pages 307–315, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[7] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[8] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[9] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 368–387, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[10] Jun Furukawa, Hiroshi Miyauchi, Kengo Mori, Satoshi Obana, and Kazue Sako. An implementation of a universally verifiable electronic voting scheme based on shuffling. In Matt Blaze, editor, *Financial Cryptography*, pages 16–30, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] Jun Furukawa. Efficient, verifiable shuffle decryption and its requirement of unlinkability. In Feng Bao, Robert Deng, and Jianying Zhou, editors, *Public Key Cryptography – PKC 2004*, pages 319–332, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[12] Josh Benaloh. Simple verifiable elections. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, EVT'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[13] James Heather and David Lundin. The append-only web bulletin board. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 242–256, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

A Proof of Shuffle

A.1 Variable mapping

The following table presents how the variable names are mapped between the main protocol of [9] and our protocol described in section 2.6.

Original paper	Our protocol	Comments
p		p is not visible in the algorithm as it is inferred in the elliptic curve operations
q	q	the curve order
g	G	the curve generator
y	Y	the encryption key
(g_i, m_i)	$e_i = (R_i, C_i)$	the original list of cryptograms
(g'_i, m'_i)	$e'_i = (R'_i, C'_i)$	the mixed list of cryptograms
σ	σ	
ρ	ρ	
τ	τ	
α	a	
α_i	α_i	
λ	λ	
λ_i	λ_i	
t	T	
v	V	
w	W	
u	U	
u_i	U_i	
g'	R'	
m'	C'	
$\dot{t}_i$	$\dot{T}_i$	
$\dot{v}_i$	$\dot{V}_i$	
$\dot{v}$	$\dot{V}$	
$\dot{w}_i$	$\dot{W}_i$	
$\dot{w}$	$\dot{W}$	
c_i	c_i	the challenges
s	z	
s_i	s_i	
λ'	λ'	