POLYAS 3.0 Verifiable E-Voting System

Version 1.3.2

July 31, 2023

Tomasz Truderung

POLYAS GmbH

Contents

List of Algorithms

1 Introduction

This document describes a variant of *POLYAS 3.0 E-Voting System* offering universal verifiability and, optionally, individual verifiability (cast-as-intended) with the use of a second voter's device (a mechanism not described in this document).

We describe the structure of the protocol and the intended security properties. We also describe the tasks assigned to the Election Council (the entity which organises and supervises the election process) in order to support some security aspects (such as universal verifiability). Further details, including in particular a detailed specification of the verification procedure, are provided in appendices.

The proposed system has the potential to provide a whole range of state-of-the-art security features, including ballot privacy and end-to-end verifiability. Some aspects of these security features require, however, participation of additional independent parties, according to the principle of division of roles; the role of such independent parties could be carried out by, for instance, entities designated by the Election Council or an organisation representing voters or candidates. While participation of such independent parties involves some additional organisational effort, this effort can be scaled depending on the required security level.

The version of *POLYAS 3.0 Verifiable* e-voting system we currently offer to our clients provides the following features (see section Section 2 for more details):

- **Universally verifiable tallying**, based on verifiable mixing and verifiable decryption, using state-of-the art cryptographic techniques, such as zero-knowledge proofs (in this instance of the system we will have one mixing node).
  The corresponding verification tools (implemented based on the specification provided in this document) can be used to verify correctness of the tallying process. A reference implementation of the verification procedure is provided by POLYAS (including the source code).
- **Eligibility verifiability.** Voters' credentials are generated by an entity designated by the Election Council. The passwords are then distributed to the voters using, for instance, a secure printing facility. This mechanism provides a measure against so called ballot stuffing (unauthorized injecting of additional ballots to the ballot box by the Election Provider).
  POLYAS provides a tool for credential generation, including the source code. Such a tool can be also independently implemented (by the Election Council or a third party), based on the this documentation.
- **Individual/cast-as-intended verifiability** based on the use of a second device by the voter (this mechanism is optional and not described in this document).

The three above features combined provide so-called *end-to-end verifiability*.

In what follows, we describe the general structure of the current configuration of *POLYAS Core 3.0 Verifiable* and discuss its intended security properties. Then we provide an overview of the verification tasks, that is tasks to be implemented in order to build a verification tool for universal verifiability of the tallying process. In the appendices we provide details on the used cryptographic modules and the protocol structure. These details should enable any interested party to independently implement a verification procedure.

2 Protocol Structure and Security Properties

In this section we describe the structure of the protocol and the division of roles. We also and enumerate security properties the system is aiming at, along with the trust assumptions under which these security properties are meant to be achieved.

2.1 Overview of the Protocol

The protocol structure of the currently offered configuration is depicted on Figure 1.

The protocol involves three roles:

- the *Election Provider* (Polyas),
- the *Registrar*,
- the *Distribution Facility*, and
- the *Election Administrators*

The roles of the Registrar, the Election Administrators, and the Distribution Facility are, typically, carried out by entities designated by the Election Council (the organisation carrying out the elections) or by a sub-entity of this organisation. The role of the Distribution Facility, carried out, for instance by a trusted printing facility, is to distribute the passwords generated by the Registrar to the voters.

The system uses standard ElGamal-based cryptography with verifiable mixing (based on the construction of Wikstroem at al.) and verifiable decryption.

In the *registration phase* (Step 1 on Figure 1), the Registrar generates voters' passwords (private credentials) and the corresponding public credentials (private voters' credentials and the corresponding public credentials are in a cryptographic relation: roughly, the private credential can serve as a signing key and the corresponding public credential is the corresponding verification key). The voters' passwords are transferred in an *encrypted form* to the secure Distribution Facility (such as printing facility) which distributes to the voters (for instance by postal mail). The public credentials of all eligible voters are uploaded by the Registrar to the voting system (POLYAS) and published on the registry board.

During the *voting phase* (Step 2), a voter opens the election web page (running the voting

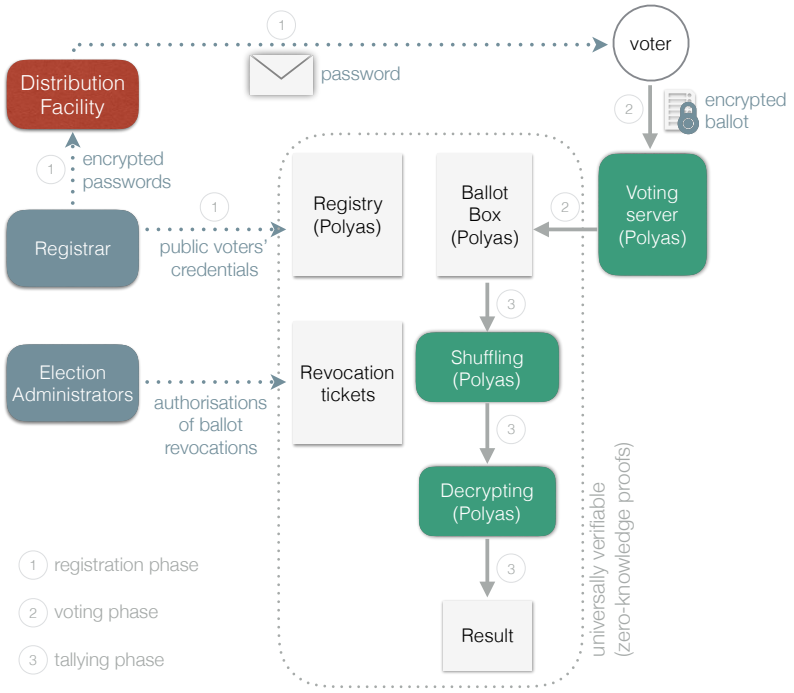

Figure 1: The structure of the system. Gray boxes represent bulletin boards. Ballot box is a bulletin board, where submitted ballots are published. Although not indicated on the picture, the intermediate results of shuffling (mixing) are also posted on bulletin boards and hence are subject to audit. In particular, verifiable shuffling and verifiable decryption produce zero-knowledge proofs of correctness of these operations.

client code in the voter's browser), authenticates himself/herself to the voting platform, using his/her voter login and password, and indicates his/her voting choices. The voting client creates an encrypted and signed ballot containing the selected choice (the ballot is signed with the voter's private key derived from his/her password) and submits this ballot to the voting server which adds the ballot to the ballot box. Note that the ballot is encrypted and signed on the client side and hence the voter's choice does not leave the voter's computer unencrypted; also the voter's private key is never transfered to the server.

The POLYAS Core 3 system can be configured to support so called *ballot revocations*, a verifiable and transparent mechanism to revoke ballots of chosen voters. If the system is so configured, the Election Administrators can create and authorise, so called, *revocation tickets*. A revocation ticket includes a list of voters whose ballots are to be revoked. The *revocation policy* of the given election specifies the minimal required number of Election Administrators who should authorise a ticket. Depending on the chosen variant, the election administrators may need to digitally sign revocation tickets in order to authorise

them. Correctly authorised revocation tickets, along with the authorisations, are published on a designated bulletin boards. Ballot cast by the voters listed in correctly authorised revocation tickets do not take part in the tally (they are filtered out before the mixing step).

In the *tallying phase* (Step 3), the encrypted ballots from the ballot box (which are not revoked, as described above) are mixed and decrypted in a verifiable way (that is, zero-knowledge proofs of correct shuffle and correct decryption are produced), which means that the tallying process is fully verifiable. As explained later, the Election Council can verify the proofs to make sure that tallying was carried out correctly.

We will now discuss the security features the system is designed to provide.

2.2 Privacy

In the variant of the POLYAS Core 3 system described on this document, POLYAS generates and maintains the private elections key and publishes the corresponding public election key (used to encrypt ballots). Therefore, POLYAS is trusted to handle and use the private election key properly (that is use this key exactly as specified by the protocol), in order to provide ballot privacy.

Note: We also offer a variant of the system using a threshold decryption scheme, where the private election key is shared between the election system and some number of independent (external) actors (trustees). These authorities participate in cryptographic shuffling and decryption.

2.3 Universal verifiability

As already mentioned, the complete tallying process (taking as its input the content of the ballot box, the registration board, and, if ballot revocations are supported by the election, the content of the revocation boards) is fully verifiable. It uses standard cryptographic methods to achieve this end, including fully verifiable shuffle and the decryption by the means of zero-knowledge proofs.

The universal verifiability mechanism discussed here allows one to check the integrity of the tallying process without any trust assumptions.

The process is organised as follows. At the end of the election, POLYAS provides the Election Council with the data packet containing the election result, along with the proof of correctness of the tallying process. More precisely, this data packet contains the content of all the public bulletin boards which include *the final content of the ballot box*, as well as the results of all the intermediate steps with appropriate zero-knowledge proofs. The Election Council can then check the proofs (using a verification tool), in order to confirm that the tallying has been done correctly (and the election result is correct with respect to the included content of the ballot box).

The bulletin boards relevant for the verification process are

- **the registry** containing, most importantly, the list of public credentials of eligible voters,
- **the ballot box** containing the final list of encrypted ballots to be tallied; the verification tool should check that all the ballots are well formed and properly signed (using secret credentials corresponding to the public credentials published in the registry),
- **the revocation boards** containing, if the election is configured to support revocations, revocation tokens and authorisations for these tokens.
- **the result of verifiable shuffling** with the zero-knowledge proofs of correct shuffle; the verification tool should check these proofs,
- **the decrypted ballots** (the final result) with the zero-knowledge proofs of correct decryption; the verification tool should check these proofs.

Note that the actual final tally can be computed based on the content of the last bulletin board.

Detailed cryptographic and technical specification of the tallying process, the intended content and data format of all the public bulletin boards, and the corresponding verification procedure, are provided in the in the Appendix. Based on this specification, independent parties can implement verification tools (which can be used by the Election Council). POLYAS also also provides a reference implementation of such a tool with the source code.

2.4 Protection against ballot stuffing

The ballot box is maintained and controlled by POLYAS which is supposed to publish there only encrypted ballots submitted by correctly authorized voters. To avoid hypothetical overuse of the control of the bulletin board (by adding additional illegitimate ballots), the system implements a protection mechanism based on the use of independently generated voter's passwords (called here also *voter's secret credentials*).

Such passwords are generated by independent parties called the *Registrar* and the *Distribution facility*: each eligible voter obtains his/her password (which is, in its essence, a private signing key), while all the corresponding public credentials (corresponding public verification keys) are uploaded by the Registrar to the election registry board. The passwords are distributed by the Distribution Facility (such as a trusted printing facility).

During the voting phase, the voter's password is used, together with his/her voter identifier, to authenticate the voter and then to sign the encrypted ballot (the signing key is derived from the password). This signature is checked, first, by the voting server (to make sure that the ballot is authorized) and later by the verification procedure (carried out by the Election Council).

We note here the following properties of this process, which provide countermeasures

against ballot stuffing:

- To create a valid ballot, the password of an eligible voter is needed. Adding a ballot which has been not signed with a valid password would be detected by the verification procedure.
- POLYAS does not have the knowledge of voter's passwords and, therefore, only ballots of those voters who vote (and provide their passwords) can be correctly created. POLYAS is therefore not able to fabricate ballots of voters who have not voted (as mentioned above, adding ballots without valid signature would be detected during the universal verifiability procedure).
  This property holds under the assumption that the distribution/printing facility does not collude with POLYAS and that password are strong enough (Appendix A.3 provides details on the password generation and how signing keys are derived).

POLYAS provides a simple application, including the source code, for generating secret/public voters' credentials. We will also provide a specification of such an application, so that it can be independently implemented by a third party.

See Section A.3 for the details of the password generation process.

Trust assumptions The system provides protection against ballot stuffing, as defined above, under the following trust assumptions:

- The Registrar is honest (does not manipulate the software for password generation in order to circumvent the cryptographic measures used by this software to protect the passwords).
- The Printing Facility is honest (does not reveal voter's passwords to any third parties, in particular to the Election Provider).

3 Overview of the Verification Tasks

In this section we describe architecture of the system in more detail, providing in particular the list of all boards to be verified, and give an overview of the tasks of the verification procedure.

Bulletin boards of the described system are depicted on Figure 2. Bulletin boards offer append-only storage facility. They provide the only way the data is transfered between the sub-components, which makes the transfer of data (the protocol transcript) transparent and explicit. Content of these boards constitutes the input to the verification procedure.

The data published on the, so-called, *external boards* (represented in orange) comes from the outside of the system: the registration data published on the registry board is generated externally by the credential generation tool, the revocation tickets and the corresponding

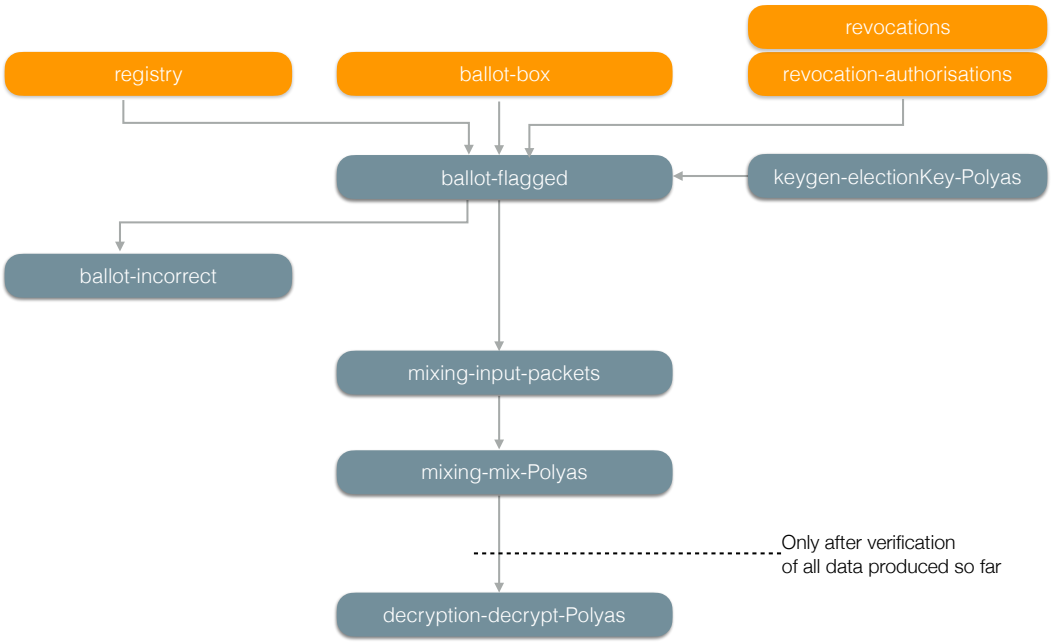

Figure 2: Bulletin boards of the system with indicated information flow between them. Orange boxes represent *external boards*; the content of all the remaining boards is computed (from the indicated input) by the Polyas Core 3.0 back-end.

aurhorisations published on the revocations and revocation-aurhorisations boards are created by election administrators, while encrypted ballots, published on the ballot-box board, are created on the client side during the voting casting process and published via the voting server. The content of the remaining (internal) boards is computed by the Polyas Core 3.0 back-end from the indicated input.

The goal of the verification process is to check that the content of all internal bulletin board has been computed correctly. Additionally one needs to make sure that the content of the external boards is as expected.

The cryptographic details of how the system works and how the verification of different boards should be implemented are given in the following sections. Here we provide an overview of the verification tasks.

Cryptographic setup and auxiliary algorithms. In order to implement the verification tool, one needs to first implement some auxiliary cryptographic algorithms. Such algorithms are described in Section A.1. This section includes, in particular, descriptions of a key derivation procedure (Algorithm 1) and an algorithm for a hashing into $\mathbb{Z}_q$ (Algorithm 4), which are building blocks of many verification algorithms.

The general cryptographic setting (the used ElGamal group and some algorithms for this group) is given in Section A.2. It includes, in particular, Algorithm 7 for deriving independent elements of the group in a reproducible (verifiable). Such independent elements (generators) are a necessary part of the proof of correct shuffle.

Registry board. The expected content of the registration board is described in Section A.3, where also the credential generation process is detailed. The verification task related to the content of this board is to simply make sure that the registration data is well-formed (as described in the mentioned section). The Election Council (who uploads the content of this board) needs to make sure that this board contains the intended content.

Election key. The election key, along with appropriate zero-knowledge proof, is generated by the Election Provider (Polyas) and published on board keygen-electionKey-Polyas (Section D.6.2)

Cryptographic details of this process are given in Section A.4. The necessary verification step is to check correctness of the zero-knowledge proof published alongside the election key as described in Section A.4 (see Algorithm 8).

Ballot box and ballot pre-processing. Ballots are encrypted on the client side and published (via the voting server) on the ballot box. The voting server is supposed to publish only well-formed encrypted ballots and only one ballot for each voter. It means that invalid ballots should never by published on the ballot box. However, to transparently handle the case where (for any unanticipated reasons) an invalid or duplicated ballot gets published, we apply so-called ballot preprocessing process where invalid and/or duplicated ballots are, first, explicitly flagged and, then filtered out. Also at this point, if the election supports ballot revocations, the revoked ballots are flagged as well.

The cryptographic details of this process are given in Section A.5. The verification task here is to check that the invalid ballots are correctly flagged. To this end, the verification procedure needs to check correctness of the zero-knowledge proofs included in the ballot entry (see Algorithm 9).

Creation of mix packages and verifiable shuffling. The ballots are shuffled in packets of a fixed maximal size K, where K is a parameter specified on the registration board. Board mixing-input-packets (Section D.7.1) contains input mix packets, where all the

eligible ballots, that is ballots not flagged as incorrect or revoked, from the ballot-flagged (Section D.2.1) board are grouped in packets respecting the following rules:

1. the order of the ballots is preserved,
2. the maximal number of elements in a packet is K,
3. the minimal number of elements in a packet is $K/2$, unless the total number n of correct ballots is smaller that $K/2$, in which case the size of the (only) packet is n.

If different voters can vote for different ballot sheets, which is reflected in the fact that different voters have assigned different *public labels* (in short, a public label specifies the list of ballot sheets a voter is eligible to vote; see Appendix A.5 for more explanation), then the above process is done independently for each public label (each packet contains encrypted ballots of voters with the same public label, the order of ballots for each public label is preserved, and so on).

The verification task for this board is to check that the packets have been created correctly, that is preserving the listed rules.

The packets of ciphertexts published on board mixing-input-packets (Section D.7.1) are then cryptographically shuffled (packet-wise) and the shuffled (and re-encrypted) packets of ciphertexts are published on board mixing-mix-Polyas (Section D.7.2). The cryptographic details of the verifiable shuffling and the corresponding verification procedure are given in Section A.6.

Verifiable decryption. The process of verifiable decryption takes, as its input, the shuffled ciphertexts and decrypts them, providing additionally zero-knowledge proof that the decryption has been carried out correctly (without revealing the secret election key). The decrypted ballots are published (in packets) on board decryption-decrypt-Polyas (Section D.4.1).

Cryptographic details are given in Section A.7. Verification of this step requires an implementation of Algorithm 13 which checks the zero-knowledge proofs of correct decryption.

We want to note that Polyas Core 3.0 back-end routinely carries out the verification of all its sub-components (including verification of the published zero-knowledge proofs) in order to make sure that the process proceeds as expected. Importantly for protecting ballot privacy, the decryption process (which decrypts the shuffled ballots) is only carried out after the content of all the preceding bulletin boards is fully verified. It means that the decryption key is used only after making sure that the input ciphertexts are exactly as expected. This is an important part of the policy of correct handling of the secret election key.

Ballot decoding and final tallying. Board decryption-decrypt-Polyas (Section D.4.1) contains decrypted ballots in binary format. These ballots are then decoded in order to

carry out the final tally (which produces final, human-readable result). The verification of this step should re-calculate the final result based on the binary decrypted ballots in order to check that the claimed result is correct. Details of this step are given in Section A.8.

Additional verification tasks. Additional verification tasks are concerned with integrity checks related to voters' receipts (signed ballot confirmations) and the set-up of the tools for cast-as-intended verifiability.

- When the election system is deployed and bootstrapped, it creates the so called *second-device public parameters*, that is parameters of the voting process relevant for the ballot receipt signing/verification and ballot audit (cast-as-intende verification with a second device). If a second device application is deployed, it should be pre-configured with this fingerprint.
  A tool for universal verifiability should offer the possibility of checking that a given file with the public parameters fingerprint is consistent with the content of the bulletin boards. This process is described in Appendix B.1.
- Voters have the option to save *ballot cast confirmations* (receipts), containing fingerprints of their encrypted ballots, signed by the election system. The voters can then hand over their receipts to auditors (who carry out the universal verifiability procedure), in order to make sure that their ballots are included in the final tally.
  To support this process, a verification tool needs to be able to process receipt files and make necessary consistency checks, as described in Appendix B.2.

Additional information. Appendix D contains technical details of all the involved bulletin boards. It includes also example data (taken from an actual, small system run). Appendix E lists and documents data types used in the system. Appendix C describes the format of the verification data packet.

A Cryptographic Modules

In this section we provide details of the cryptographic building blocks used in the system.

A.1 Auxiliary Algorithms

A.1.1 KDF

In this section we describe a key-derivation algorithm, used to generate pseudo-random byte arrays from a given seed. Our implementation follows algorithm 5.1 from KDF in Counter Mode (NIST SP 800-108).

Based on this algorithm, we then define procedures that generate (from the given seed and in a reproducible way) numbers in the given range.

The key derivation (KDF) procedure. To generate the pseudo-random bytes from the given seed (and some additional, so called, domain parameters) we use Algorithm 1. This algorithm depends on a pseudo-random function. Following the NIST recommendation (which suggests using HMAC or CMAC), we use HMAC-SHA512 as our pseudo-random function (PRF), which we denote by *mac*.

Algorithm 1: Key derivation function

Input :

- the desired length l (in bytes) of the pseudo-random output
- the initial seed k (so called *key derivation key*), given as a byte array,
- a byte array *label* that identifies the purpose of the derived key material
- a byte array *context* that, again, describes the context of the derivation procedure.

Output: Sequence of l bytes computed in the following way:
Denote by B_i the block of 64 bytes (512 bits) computed by:

$$B_i = mac_k(i \,||\, label \,||\, 0x00 \,||\, context \,||\, l)$$

where i and l above are represented as a 4-byte integer. (The NIST standard we refer to uses at this point the desired length *in bits*; here we define the procedure to operate on the byte-size granularity level.)
To produce the output, compute the necessary number of blocks $B_0, B_1, \ldots$ and take the first l bytes of this sequence.

Example: For the initial seed set to `"kdk"` (more precisely, to the byte representation of this string using the UTF-8 encoding; we will use this convention throughout the following examples), the label set to `"label"`, context set to `"context"`, and the desired byte length l set to 65, the above algorithm produces the following byte string (in the hexadecimal representation):

```
32 88 92 2A 96 65 33 C7 93 ED 53 20 45 FF FC 3C E6 BA 77 F2 7E 8F 60 C9 A3 D8 22 21 D8 6F 51 DD A0 07 36 DB
A3 F8 AE 1D 94 B1 75 62 E8 38 D5 7F B8 54 00 D1 47 C6 E9 58 5E D4 D8 59 E4 61 20 B2 75
```

Numbers from seed. We describe now a method which produces a sequence of positive integers of a given bit length, generated deterministically based on the given seed and index, following algorithm A.2.3 Verifiable Canonical Generation of the Generator g from NIST fips186-4 (this method implements, in essence, a part of this algorithm). This method is used, in particular, to verifiable produce generators, as we will explain in the next section. The method is described in Algorithm 2.

Example: The first three numbers generated using Algorithm 2 for the initial seed `"xyz"`

Algorithm 2: Numbers from seed

Input :
- the desired length l in bits of the generated numbers,
- the initial seed *seed*, given as a byte array,

Output: Sequence $n_1, n_2, \ldots$ of numbers in the range $[0, 2^l)$; the maximal length of the output sequence is limited by `MAXINT32` which is the maximal number that can be represented as a 32-bit signed integer

1 **for** $i \in 1, \ldots$ **do**
2 Derive $\lceil l/8 \rceil$ bytes, using Algorithm 1, for the initial seed $k = seed \| i$, with i represented as 4-byte integer, *label* being set to the byte representation of the string "generator", and *context* being set to the byte representation of the string "Polyas".
3 Ignore the appropriate number of left-most bits from the byte array above, to obtain a bit array t_i of length l.
4 Define n_i as the positive integer with the bit representation t_i. (*Implementation note: in some implementations, for example if the the standard Java* `BigInteger` *class is used, one may need to add some leading zeros to the byte sequence in order to guarantee that the bit array will not be interpreted as a negative integer*).

and $l = 520$ are

a_1 = 1732501504205220402900929820446308723705652945081825598593993913145942097001127020633138020218038968109094917857329663184563374015879596834703721749398989648

a_2 = 2207401303665503434031531355511922974889692817601183500259263742625914061046146142929376778072827450461936300533206904979740474482058840003720379960491023511

a_3 = 1883889587903519477357838514223953979954201344665681798367023196328721975720052153913582122151913785273222921786889836987731296728825119604809609410157987402

a_4 = 1423259849467217711185874799515607842842602785767879766623736284680209832704638390900412597196948750015976271793930713744890547611655064835165883233889981463

The above algorithm can be easily adopted to generate a sequence of numbers from a given range $[0, b)$, as specified in Algorithm 3.

Algorithm 3: Numbers from seed (range)

Input : Positive integer b and an initial seed *seed* given as a byte array

Output: Sequence $n_1, n_2, \ldots$ of numbers in the range $[0, b)$

1 Use Algorithm 2, with the initial seed *seed*, to generate a sequence of positive integers of the bit length l, with $l = \lceil \log_2(b) \rceil$.
2 Discard all the elements which are not in the desired range $[0, b)$.

Example: The first two numbers generated using Algorithm 3, for the initial seed "xyz"

and b set to $a_1 + 1$ (where a_1, a_2, a_3, a_4 are as in the previous example) are a_1 and a_4 (note that a_1 and a_2 are not included).

A.1.2 Hashing into $\mathbb{Z}_q$

In several cryptographic constructions involving ElGamal groups, we need a hash function which, for given input data, returns an element of $\mathbb{Z}_q$ (for some number q). To this end, we use the SHA-512 algorithm as the building block and follow the following conventions.

The **input data** (if not directly given as a byte array) before being fed into SHA-512, is transformed into a byte array in the canonical way. In particular:

- Strings are transformed to a byte array using the "UTF-8" encoding.
- Integers, if not explicitly stated otherwise, are represented as 4 bytes in the big-endian byte order (with the most significant byte coming first).
- Big integers represented by Java class `BigInteger` are transformed to byte array using the following method. One takes the byte representation of b of the given big integer, as obtained by calling method `toByteArray()`, and prepends it with 4 bytes representing the length of b.
  For instance, the number 98162874527223464716009286152 gets transformed to byte array `00 00 00 0d 01 3D 2E 6D 3A FD EC 0F 0A 00 AD 2A 08`.
- Elliptic curve points are transformed into a byte array using the compressed form of ANSI X9.62 4.3.6.[1]
  For instance, the point

  (`75788b8a22a04baad44c66ec80e86928597979bf1b287760ad4e3153293d613b`,
  `664663757d16eff0b993ac12a1ba16ee4784ac08206b12be50f4d954d9d74c88`)

  (of the `secp256k1` curve) gets transformed to byte array
  `02 75 78 8B 8A 22 A0 4B AA D4 4C 66 EC 80 E8 69 28 59 79 79 BF 1B 28 77 60 AD 4E 31 53 29 3D 61 3B`
- For an ElGamal ciphertext (x, y) (as for pairs in general), we first digest the first component x and then the second component y.

Note that the order of the input elements is significant: the elements must be digested always in the specified order.

Considering hashing into $\mathbb{Z}_q$, we distinguish two cases:

- **Non-uniform hash.** If it is not relevant that the distribution be uniform, we apply the SHA-512 function to the input data (using the described above conventions), obtaining the byte array h and then transform it into a number by constructing a Java big integer object directly from h and modulo q: `(new BigInteger(`h`)).mod(`q`)`.

[1] Or equivalently http://www.secg.org/sec1-v2.pdf in section 2.3.3. In the case the BouncyCastle library is used, this is achieved for an elliptic curve point `p`, by calling `p.getEncoded(true)`.

- **Uniform hash.** This method, presented in Algorithm 4, distributes the result of the hash function (pseudo)-uniformly in $\mathbb{Z}_q$.

Algorithm 4: Uniform Hash

Input : Positive integer q and input data to be digested
Output : A number in the range $[0, q)$

1 Apply the SHA-512 function to the input data (using the above conventions), obtaining a byte array h.
2 Use h as the seed to Algorithm 3 with parameter $b = q$ and
3 **return** the first number of the sequence generated by this algorithm.

We will denote application of Algorithm 4 with parameter q to some data $a_1, \ldots, a_n$, where a_i are elements that can be converted to byte arrays, using the conventions described above, by $\mathit{uniformHash}_q(a_1, \ldots, a_n)$.

Example: The uniform hash algorithm, for $q = 2126991829$ and input data `"some data"` returns 414907466.

Example: For $q = 2126991829$, $s =$ `"some data"`, and $n = 98162874527223464716009286152$ (where n is of type `BigInteger`), the result of $\mathit{uniformHash}_q(s, n)$ is 1444258901.

A.1.3 Encoding Plaintexts as Numbers

Before a given plaintext messages (byte array) *msg* can be encrypted using the ElGamal encryption scheme, it needs to be represented as a sequence $a_1, \ldots, a_n$ of integers in the range $[0, q)$ (note that in the general case, it may be not possible to represent the plaintext message as a single integer in this range), as detailed below. Such a sequence is called a *multi-plaintext* (as it consists of possibly several simple plaintexts $a_1, \ldots, a_n$).

Such an encoding is presented in Algorithm 5. We note that this algorithm is presented here for completeness and does not have to be implemented for the verification task which only requires the corresponding decoding algorithm presented next.

The reverse operation—that is decoding a given multi-plaintext back as a message—is presented in Algorithm 6.

Example. For $q = 2^{32} - 1$ and the message *msg* = `"qwertyuioplkjhgfdsazxcvbnm"` (that is the byte representation of this string), Algorithm 5 returns multi-plaintext

625, 7824754, 7633269, 6909808, 7105386, 6842214, 6583137, 8026211, 7758446, 7143424

This multi-plaintext is mapped by Algorithm 6 back to *msg*.

Algorithm 5: Encoding a message as a multi-plaintext

Input : A positive number q and a byte array msg with consecutive bytes $b_1, \ldots, b_l$

Output : Encoding of msg as a multi-plaintext $a_1, \ldots, a_m$

1 Compute the block size $s = \left\lfloor \frac{\log_2(q)}{8} \right\rfloor$. (This value expresses how many bytes can be represented as one number in the range $[0, q)$; more precisely, s bytes can by represented as a number in the range $[0, u)$, where $u = 2^{8s}$; note that $u \leq q$.)

2 Compute a padded version msg' of msg by prepending it by two bytes containing the length of the pad and appending the pad, where the pad consists of zero-bytes and the length k of the pad is the minimal integer such that the size of msg' is dividable by the block size s. That is, msg' consists of bytes

$$c, c', b_1, \ldots, b_l, z_1, \ldots, z_k$$

where each z_i (for $i \in \{1, \ldots, k\}$) is a zero byte, the bytes c, c' contain the value k (the pad length encoded in two bytes using the *big endian* byte order (where the most significant byte is in the left-most one), and the length of the pad $k = \left\lceil \frac{l+2}{s} \right\rceil \cdot s - (l + 2)$. Note that the length of msg' is $\left\lceil \frac{l+2}{s} \right\rceil \cdot s$ which is indeed dividable by s.

3 Repeat the following step: take s consecutive bytes of msg' and convert them to a positive integer (such an integer will be in the range $[0, q)$ by the definition of s), using the *big-endian* byte-order. Note that, for some implementations one needs to be make sure that the leading bit is zero, in order to obtain positive integers. By repeating this step we obtain consecutive elements a_i.

Algorithm 6: Decoding a message from a multi-plaintext

Input : A positive number q and a multi-plaintext $a_1, \ldots, a_m$, where $a_i \in [0, q)$

Output : A byte array msg

1 For each number a_i obtain its representation as a byte array of length s, where s is like in Step 1 of Algorithm 5. Let msg' be the concatenation of the byte arrays obtained in the above way for all the numbers from a_1 to a_m.

2 Interpret the first two bytes of msg' a positive integer k.

3 Check that k right-most bytes of msg' are zeros.

4 Truncate two left-most bytes and k right-most bytes of msg' to obtain msg.

5 **return** msg.

A.2 Used ElGamal Group

In the variant of the Polyas system described in this documentation, we use the ElGamal Group based on the elliptic curve `secp256k1` [2]. This curve is defined over the finite field F_p with

$$p = \texttt{fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f}$$

($p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$) and has coefficients $a = 0$ and $b = 7$. The generator of the group is

$$g = (\texttt{79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798},$$
$$\texttt{483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8})$$

This group is of prime order

$$q = \texttt{fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141}$$

A.2.1 Elliptic curve encoding

In order to encrypt plaintexts, which are represented as numbers in $\mathbb{Z}_q$, using the ElGamal encryption scheme over the given cyclic group, one first needs to convert the given plaintext into an element of the group; conversely, decryption requires converting a group element back into $\mathbb{Z}_q$. However, unlike the other common groups used in cryptography, in general there exists no efficiently computable homomorphism between the integers modulo q and an elliptic curve of order q.

We use therefore (in a slightly modified way) a method suggested by Neal Koblitz in Elliptic Curve Cryptosystems [7]. We note that a very similar method is used in the *Verificatum Core Routines*.

The method will use a constant k which, by default, we set to 80. Let $\mathit{messageUpperBound} = \lfloor p/k \rfloor$ (recall that p is the order of the underlying finite field).

The encoding works for positive integers $a < \mathit{messageUpperBound}$ and succeeds with overwhelming probability in k. Notice that this encoding is somehow sub-optimal, because the message space is limited to u rather than the order of the group q, but it is not of practical significance.

Encoding: Given a number a to be encoded, iterating a counter i from 1 to k, we set the x-coordinate to be $x_i = a * k + i \bmod p$. We then solve the elliptic curve equation in the standard way (using the Tonelli-Shanks algorithm to compute square roots modulo a prime) to compute y_i (if such a solution exists) such that the point (x_i, y_i) is on the curve. If so, we return this point; otherwise we continue to iterate. If after iterating k times no solution is found, the algorithm fails.

Under the (commonly accepted) assumption that the x-coordinates of the group elements are uniformly distributed in the underlying finite field, the probability of failing to encode a given number a is approximately $\left(1 - \frac{q}{2p}\right)^k$.

Decoding: Given an elliptic curve point (x, y), we compute the message $a = \lfloor (x-1)/k \rfloor$.

Example: For

a = 723700557733226221397318656304299424082937404160253525246609900049430216698

the above encoding yields the point

(`7fffffffffffffffffffffffffffffffffffffffffffffffffffffff7ffffe21`,
`2af4d53f09f4d4ede3caf3f0e06ccfc0f55289d83fed859ca504d6033bec629b`)

that is the point with the x-coordinate $x = a * 80 + 1$.

A.2.2 Select Generator Verifiably

We describe now a method for generating group elements from a given seed, inspired by Algorithm A.2.3 *Verifiable Canonical Generation of the Generator* from NIST fips186-4. When used with a (pseudo-)random seed, this method produces pseudo-random group element distributed (pseudo) uniformly in the group. When used with a pre-agreed seed, this method can be used to select independent generators of the given group in a verifiable (reproducible) way. The latter is crucial, in particular, for integrity of verifiable shuffling.

The mentioned standard covers the case of the traditional Schnorr groups (of quadratic residues modulo a safe prime); the method presented here is an adaptation of this algorithm for the elliptic-curve-based groups of prime order.

The method is presented in Algorithm 7.

Example: The numbers returned by this algorithm for the `secp256k1` curve with *seed* set to the (byte representation of the string) "seed", and for *index* in the range $1, \ldots, 3$ are:

(`879f580dfe31c74dc2b4289f1988e581c76e625761a863971c808e90ab6fd3c7`,
`4c59d9061d35678d06c04fe9f61dd47d7ee9e35b9847e5f3f9ed532c509afc0f`),
(`b6413eb866319a631509ad0e637ec260507383d7495ef66858f9a6a4bb8efac7`,
`adb88f8cdd62aad64e2518d383b4e6aa2013910964b6423c17c0100f96118ae1`)
(`1845cc619ec1a70c743e6559938290b7dac3d63b3fd2cf8d6e0646d292a576e8`,
`439f7d97af4396e0441b7d292045cf78bc22187eec981e5d6fe2ebd0794f975a`)

A.3 Credential Generation and Voters Authentication

In this section, we describe the process of generating voters' credentials (passwords) and how these credentials are distributed and used to authenticate voters and sign their ballots.

Algorithm 7: Independent generators for EC groups of prime order

Input :

- the group parameters: the prime order of the underlying field p, coefficients a and b of the elliptic curve,
- the seed *seed*,
- an integer *index* which specifies the index of the generator to be obtained from that seed.

Output : The *index*-th generator for the given parameters.

Let $w_1, w_2, \ldots$ be the sequence of numbers in the range $[0, 2p)$ generated from the seed $s = (seed||\text{"ggen"}||index)$, where *index* is represented as 4-byte integer, using Algorithm 3.

for w *in* $w_1, w_2, \ldots$ **do**

  Define $x := w \mod p$.

  Try to compute (using for instance the Tonelli–Shanks algorithm), the square root $\hat{y} = \sqrt{x^3 + ax + b} \mod p$.

  if *the square root* $\hat{y}$ *exists* **then**

    Take $y = (-\hat{y} \mod p)$, if $w < p$, and $y = \hat{y}$ otherwise (so we use the most significant bit of w to determine which of the two possible y-coordinates should be taken).

    If the obtained point (x, y) is a generator of the group (which holds always except for the point at infinity), then **return** this point (otherwise, continue to the next number w).

We describe here the variant, where the the credential generation is carried out by an (independent) entity designated by the Election Council, called the *registrar*. As a result of this process, two files are produced: one for the printing facility (which then distributes passwords to voters) and one for the election provider (Polyas), containing the public registry (with public voter's keys) and data used in the voters authentication process.

POLYAS Core3 system can be also configured in such a way that the same algorithm for password generation (as described below) is carried out automatically by Polyas during the setup of the election. In this case, the passwords are delivered to the voters directly (via emal) by Polyas.

Cryptographic setting and tools. We assume that the public election parameters include a specification of an ElGamal group G of order q with a generator g. The method described in this section works for any group, where the DLOG problem is hard, but in the specific instantiation we have implemnted, the chosen group is the elliptic curve group `secp256k1` (see Section A.2).

We will use a hash function H and a key derivation function $KDF : \mathit{String} \to \mathbb{Z}_q \times \mathbb{Z}_q$, which from a given input string produces two (pseudo-uniformly distributed) integers in $\mathbb{Z}_q$.

Specifically, in our instantiation, by H we denote the `PBKDF2` function (Password-Based Key Derivation Function 2) with `SHA512` [1]. We choose the following instantiation for the KDF: for the given in input string s, the result (a, b) of KDF is computed as follows:

$$a = \mathit{uniformHash}_q(s)$$
$$b = \mathit{uniformHash}_q(\texttt{"derive-password"} \| s)$$

where the symbol $\|$ denotes byte string concatenation.

Credential Generation. The process of credential generation takes, as its input, the following data:

- a list $(id_1, address_1), \ldots, (id_n, address_n)$ of voter identifiers and their addresses (these addresses will be used by the printing facility to deliver credentials to voters),
- the public encryption key of the printing facility.

Credential generation proceeds as follows. First, for every voter i, a random password p_i is generated. It will be later delivered to the voter by the printing facility. Such a password is a string consisting of alphanumerical characters (from a variant of the Base32 character set[2]) containing, in the recommended variant, between 80 and 100 bits of entropy, depending

[2]The characters used in passwords are: `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `J`, `K`, `L`, `M`, `N`, `P`, `Q`, `R`, `S`, `T`, `U`, `V`, `W`, `X`, `Y`, `Z`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`. This set does not contain pairs of similarly looking (and hence confusing) characters such like `0` and `O`.

on the desired security level. An example password, containing 80 bits of entropy, may look as follows.

```
HLGN G4QM 6M36 SL7T
```

Next, the key derivation function is applied to the password concatenated with the voter identifier:

$$(sk_i, sk'_i) = KDF(p_i \| id_i).$$

which gives values $sk_i, sk'_i \in \mathbb{Z}_q$.

Finally, the following values are computed:

$$\begin{aligned} pk_i &= g^{sk_i}, \\ dp_i &= g^{sk'_i}, \\ h_i &= H(dp_i) \end{aligned}$$

where H is a hashing algorithm with salt (in our instantiation, to compute $H(m)$, a random salt s is generated, $h = SHA256(s\|m)$ is computed, and the algorithm returns the pair (s, h)).

The intended use of these values is as follows:

- sk_i is the *private signing key* of the i-th voter which will be derived from the voter's password (by the voter's client application, using the *KDF* as above) and used to sign ballots; pk is the corresponding *public key* which will be uploaded to the voting server as part of the public registry and used to validate the voter's ballot.
- h_i, the *hashed password*, will be uploaded to the election server and used to authenticate the voter in the following way. The voter will provide to his/her client application his/her voter identifier and password p_i from which the *derived password* dp_i will be computed and sent to the server; the server will hash this derived password and match it against the stored hash h_i.
  The (somehow indirect) way the hashed passwords are generated guarantees that, given the hashes, it is not easy to brute-force voters' passwords: such brute forcing is by construction exactly as hard as brute forcing the voters' passwords, given their public credentials.

Having computed the above values for each voter, the credential generation process will produce the following output files, one for the printing facility and one for the election provider (Polyas):

- A **file intended for the printing facility**, containing records of the form $p_i,\ address_i$ (that is the voter's password and address). This file is encrypted using the public encryption key of the printing facility.
  The file is intended to be used in the following way. The printing facility is supposed to print the passwords and deliver them to the appropriate voters (with the given addresses).
- A **file intended for the election provider**, with records containing the voter's identifier (id_i), the hashed password (h_i), and the voter's public credential (pk_i).
  The file is intended to be used in the following way. The list of public keys will be published on a registry bulletin board (as such, they will be then used in the universal verifiability procedure), while the remaining fields (id_i and h_i) will be used for the voters' authentication (these fields will not be published).[3]

The tool can handle additional fields in the input data records. These fields can be appended to both output files. Importantly, however, voter identifiers are not included in the file for the printing facility and, on the other hand side, passwrods are not included in the file for the election provider.

A.4 Key Generation

In this section, we are concerned with the simple key generation process, where the election key is generated by one authority (POLYAS). The variant of our election system with key sharing and threshold decryption is not covered by this document.

The secret (decryption) election key and the corresponding public (encryption) election key are generated by the Election Provider (Polyas). The secret key s is a random number in $\mathbb{Z}_q$. The corresponding public election key pk is derived from the private key in the standard way:

$$pk = g^s$$

and published, along with a zero-knowledge proof π of the knowledge of the corresponding secret key. The public election key used, in particular, by the voter client to encrypt the voter's ballot.

The zero-knowledge proof π of knowledge of the secret key corresponding to the given public key pk is the non-interactive variant of the well-known zero-knowledge proof of the knowledge of the discrete logarithm of pk [8], where we use the strong Fiat-Shamir heuristic [3]. It is of the form $\pi = (c, f)$ and is created by drawing a random number a

[3] We are describing here a variant where the verification package, for the sake of voter anonymity and additional voters' privacy, does not contain the (real-life) voter identifiers. In this case, on the bulletin boards and in the verification data package, the voters are consistently identified by their public credentials which are randomly created by the process described here (therefore, as far as the bulletin boards and the verification data package are concerned, public credentials serve as unique voter identifiers).

from $\mathbb{Z}_q$ and computing

$$\begin{aligned} A &= g^a \\ c &= \mathit{UniformHash}_q(g, pk, A) \\ f &= a + cs \mod q \end{aligned}$$

where $\mathit{UniformHash}_q$ denotes the uniform hash function defined in Algorithm 4 with parameter 'q' set to q (the order of the group).

Note. *A record containing the public election key along with the corresponding zero-knowledge proof is published on board keygen-electionKey-Polyas (Section D.6.2). Such a record is of the type PublicKey-WithZKP (Section E.35).*

The public output of the key generation process (the public election key with the mentioned zero-knowledge proof) should be checked using the verification procedure described in Algorithm 8.

Algorithm 8: Verification of the public election key with the ZK-proof

Input: The public election key $pk \in G$ and the zero-knowledge proof of the knowledge of the corresponding secret key consisting of $c \in \mathbb{Z}_q$ and $f \in \mathbb{Z}_q$

1 Compute $c' = \mathit{uniformHash}_q(g, pk, \frac{g^f}{pk^c})$, where $\mathit{uniformHash}_q$ denotes the uniform hash function defined in Algorithm 4 with parameter 'q' set to q (the order of the group). Note that the computed value is an element of $\mathbb{Z}_q$.

2 Check that $c = c'$ and reject the proof as incorrect if this is not the case.

Example: For the group `secp256k` and the public key

$$pk = \texttt{03403091F3E81EE0E125FC33614DBA1ADBA569A3F7C05F9B36587054151508D490}$$

The following proof (in JSON format)

```
{
  "c" : "623279416854868254491349971992896696842074651475654801405326346508654722771 54",
  "f" : "519626871623585287094092584076364651783882473066691529650126972661198031185 83"
}
```

is a valid zero-knowledge proof of knowledge of the private key, for which Algorithm 8 succeeds.

A.5 Ballot Creation, Validation, and Revocation

In this section, we describe the intended way in which ballots are created and how they should be verified.

Ballots are encrypted on the client side, submitted via the voting server, and published in the ballot box. The voting server is supposed to publish only well-formed encrypted ballots (that is ballots with valid zero-knowledge proofs, as specified below) and only one ballot for each voter. It means that invalid ballots should never be published on the ballot box. However, to transparently handle the case where (for any unanticipated reasons) an invalid or duplicated ballot gets published, the Polyas Core 3.0 back-end first applies so called *ballot preprocessing* process where invalid ballots are, first, explicitly flagged; only the ballots which are not flagged as incorrect are then taken for the further tallying. We emphasise that, for transparency, ballots are never silently removed from the ballot box, but, instead, incorrect ballots are explicitly handled as explained.

If the election supports ballot revocations, the revoked ballots (that is ballots of voters for which appropriately authorised revocation tickets have been published) are also flagged and do not take part in the further tallying.

Note. *Ballots are published on board ballot-box (Section D.1.3) in packets of type BallotPackage (Section E.6). Each such a packet contains some number of individual ballot entries of type BallotEntry (Section E.5) which (in the subcomponent Ballot (Section E.4)) contain an encrypted ballots with the proof of knowledge of the encrypted coins and the proof of knowledge of the private credential. The ballots annotated during the ballot preprocessing, that is all the ballots enriched with the information related to the ballot status (correct/incorrect/revoked), are published (again in packets) on board ballot-flagged (Section D.2.1). The sizes of the packets published on boards ballot-box and ballot-flagged may vary and are not specified. It is only required that all the ballots from the ballot box are processed and included in board ballot-flagged in the same order (but possibly packaged in different way.)*

As mentioned above, only the ballots which are marked as correct will be further tallied. Moreover, for each ballot marked as invalid or revoked, an entry is published on board ballot-filtered-out (Section D.2.2) to explicitly protocol such an event.

Note. *In order to verify the correctness of the ballot, as specified below, one needs to read the content of the board registry (Section D.1.2), containing a record of type Registry (Section E.36). Such a record contains public election parameters including the specification of the ballot sheets and the list of eligible voters, where every voter has assigned a public credential and a public label (see below) specifying which ballot sheets the voter is eligible to vote for.*

Note. *If ballot revocations are supported by the election, the revocation policy is included in the Registry (Section E.36) record, in the field 'revocationPolicy', published on board registry (Section D.1.2). If the field 'revocationPolicy' is null, the revocations are not supported by the election. The revocation policy is given as a record of type RevocationPolicy (Section E.37) and it specifies the list of public keys of the election administrators along with the minimal number of administrators who have to authorised a revocation token (by signing it) in order for such a token to become valid. The (unauthorised) revocation tokens are published on board revocations (Section D.1.5), while the aurhorisations are published on board revocation-authorisations (Section D.1.4).*

A note on voter anonymisation. The list of voters in the registry, contains records of type Voter (Section E.48) which includes a field named `voterId` (additional fields of this record contain voter's public label (see below) and voter's public credential). This field does not, necessarily, contain the real-life voter identifier (the only assumption here is that it contains a unique string for each voter). In fact, in the default case, where the credential generation tool is used with the default settings, the voter identifiers are anonymised, that is the field `voterId` contains the same value as field 'cred' (the voter's public credential). So, in this default case, the public credential serves also as the unique voter identifier.

Note also that a ballot entry (see above) identifies a voter using only his/her public credential and not directly referring to the voter's identifier.

Public labels. Ballots published on the ballot box contain a special field called a *public label*, see BallotEntry (Section E.5), field `publicLabel`. A public label contains the public information about the ballot, namely *which ballots a voter is eligible to vote*. For instance, a public label `"A1:A3:B2"` means that the given voter is eligible to vote for three ballot sheets with identifiers `A1`, `A3`, and `B2` and, therefore, a ballot with such a label is supposed to contain encrypted votes for these three ballot sheets.

A public label for a voter (and hence which ballot sheets the voter is eligible to vote for) is specified on the registry board.

A public label is not only attached to the ballot in the ballot box, but it is propagated throughout the whole system. For instance, mix packets, that is packets of ballots to be shuffled in a verifiable way, group ballots with the same public label (the same ballot sheets). Finally, the public label is propagated to the final board containing decrypted ballots, where it provides an important context: to properly decode and tally decrypted ballots, one needs to know which ballot sheets are encoded in a given ballot; the public label provides this information.

If, in a given election, all voters vote for the same ballot sheets, then all the ballots will be labeled with the same public label.

The verification task is supposed to check that the flagging process, as introduced above, is done correctly (and report any incorrect entries). A ballot should be **flagged as incorrect** in the following cases:

1. The voter is not registered, i.e. the voter's public credential included in the ballot is not listed in the registry.
2. The public label included in the ballot is not the same as the public label of the voter specified in the registry. Note that, in the ballot entry, the voter is identified by his/her public credential.

3. The length of the encrypted choice in the ballot (i.e. the number of elements of the multi-ciphertext) is not as expected.

   To determine the expected length, one should (1) take the expected length of the encoded plaintext and (2) compute the lengh of resulting ciphertext.

   In order to compute the expected length of the encoded plaintext (Step (1) above), the following information is used: (a) the public label of the voter, as specified in the registry, which specifies identifiers of ballot sheets ($b_1, \ldots, b_l$) the voter votes for and (b) the specification of the ballots, also included in the registry board. Given this information, one can compute the expected length of an ecoded plaintext which contains choices for the ballots $b_1, \ldots, b_l$. The algorithm of ballot encoding is presented in Appendix A.8. As one can see, the expected length of the encoded plaintext is derived from the ballot structure (the number of voting options) in a straigthforward way.

   In order to compute the lenthg of the resulting ciphertexts (Step (2) above), one can construct any plaintext *msg* of the length k determined in Step (1) (for instance a plaintext *msg* containd k zero bytes) and take the length of the *multi-plaintext* computed from *msg* (see below).

4. There has been a ballot with the same public credential published before.

5. The ballot is not well formed, i.e. does not have valid zero-knowledge proofs, as detailed below.

Note. *For the last check, the public election key is used. This key is can be read from board keygen-electionKey-Polyas (Section D.6.2).*

In the next step, if the ballot revocation is enabled, the revoked ballots are flagged as such. This is applied only to the ballots which have not been already flagged as incorrect. A ballot should be **flagged as revoked**, if the voter identifier included in the ballot has been listed in at least one correctly aurhorised revocation token. The procedure of checking revocation tokens and authorisation is detailed below.

Note. *For this check, one needs to use the content of the following bulletin boards: registry (Section D.1.2), revocations (Section D.1.5), and revocation-authorisations (Section D.1.4).*

Ballot encryption. In order to create an encrypted ballot the following input is used:

- the plaintext voter's choice *msg*, given as a byte array (such a plaintext choice is created by encoding the voter's choice using algorithm presented in Appendix A.8),
- the election public key $pk \in G$ (published on the board keygen-electionKey-Polyas, Section D.6.2),
- the public label l of the voter, given as string (recall that a public label is a public information assigned to the voter which specifies which ballot sheets a voter votes for),

- the private credential (private signing key) of the voter.

This procedure uses also (implicitly) the fixed ElGamal group G of order q with a fixed generator g, as introduced above.

In the first step, the plaintext message *msg* is represented as a sequence $a = a_1, \ldots, a_n$ of integers in the range $[0, \mathit{messageUpperBound})$, so-called *multi-plaintext* (note that in the general case, it may be not possible to represent the plaintext message as a single integer in this range), using Algorithm 5 with parameter 'q' set to *messageUpperBound*.

Having computed the multi-plaintext $a = a_1, \ldots, a_n$ as described above, we encrypt every simple plaintext a_i of this multi-plaintext, using the standard ElGamal encryption, obtaining a, so-called, *multi-ciphertext* $e = e_1, \ldots e_n$, with

$$e_i = (g^{r_i}, \gamma(a_i) \cdot pk^{r_i})$$

where r_i are freshly generated random encryption coins from the set Z_q and $\gamma(a_i)$ denotes the encoding of a_i as an element of the group G, as specified in Section A.2.1. We will refer to the multi-ciphertext e as the *encrypted choice*.

Next, we compute *the proof of the knowledge of the encryption coins*, which is the non-interactive version of the well-known zero-knowledge proof of knowledge of the discrete logarithm (r_i) of the first component of each e_i [8]. This proof is of the form $(c_1, f_1), \ldots, (c_n, f_n)$ and is generated, for each $i \in \{1, \ldots, n\}$, by drawing a random element b_i from the set $\mathbb{Z}_q$ and computing

$$\begin{aligned} B_i &= g^{b_i} \\ c_i &= \mathit{UniformHash}_q(g, pk, l, e_1, \ldots, e_n, z, B_i) \\ f_i &= b_i + c_i r_i \mod q \end{aligned}$$

where z is the public credential of the voter and $\mathit{UniformHash}_q$ denotes the uniform hash function defined in Algorithm 4 with parameter 'q' set to q (the order of the group).

Finally, the ballot creation procedure computes *a proof of knowledge of the voter's signing key* bound to the encrypted choice which, essentially, is a Shnorr signature on the encrypted choice. This proof is, again, of the form (c, f) and, similarly to the above, is generated by drawing a random element b from $\mathbb{Z}_q$ and computing

$$\begin{aligned} B &= g^{b} \\ c &= \mathit{UniformHash}_q(g, pk, l, e_1, \ldots, e_n, z, B) \\ f &= b + cs \mod q \end{aligned}$$

where s is the private credential of the voter. Note that, indeed, this is a zero-knowledge proof of knowledge of the discrete logarithm s of the public voter's credential (g^s) bound to the given ballot (the content of the ballot is included under the hash).

Algorithm 9: Verification of a ballot

Input:

- public parameters: parameters of the used group G, including the generator g, and the public election key $pk \in G$,
- the voter's public label l,
- the voter's public credential (voter's public key) $z \in G$,
- encrypted choice (multi ciphertext) $e = e_1, \ldots e_n$ with $e_i = (x_i, y_i) \in G^2$ (note that (x_i, y_i) is a *pair* of elliptic curve points, not a single elliptic curve point),
- proof of knowledge of the private credential (c, f) with $c, f \in \mathbb{Z}_q$,
- proofs of knowledge of encryption coins $(c_1, f_1), \ldots (c_n, f_n)$ with $c_i, f_i \in \mathbb{Z}_q$.

1 Check that all the (sub)components of the input are in the expected domains.

2 Compute $c' = \mathit{UniformHash}_q\left(g, pk, l, e_1, \ldots, e_n, z, \frac{g^f}{z^c}\right)$, where $\mathit{UniformHash}_q$ denotes the uniform hash function defined in Algorithm 4 with parameter 'q' set to q (the order of the group). Note that, since g and z are group elements and, hence, the exponentiation and division operations denote the appropriate group operations.

3 Check that $c = c'$ and reject the ballot as incorrect if this is not the case.

4 **for** $i \in \{1, \ldots, n\}$ **do**

5 Compute $c'_i = \mathit{UniformHash}_q\left(g, pk, l, e_1, \ldots, e_n, z, \frac{g^{f_i}}{x_i{}^{c_i}}\right)$.

6 Check that $c_i = c'_i$ and reject the ballot as incorrect if this is not the case.

Correctness of a ballot entry can be checked using the verification procedure presented in Algorithm 9.

Example: In this example, we use the group `secp256k` as defined in Section A.2 and represent elliptic points in the hexadecimal representation of the the compressed form of ANSI X9.62 4.3.6.

For the public label `"a"`, public election key

$$pk = \texttt{0323863C357CF3CDFF282CB747CB23F94CCC9173B795412E773F908CC8B81AA354}$$

and the voters public credential

$$z = \texttt{03D0D99E7CB4330B6037CFC64139298DD46417D1B44781A0381CB0313F26541870}$$

the following record in JSON format is a valid ballot record for which the verification procedure should succeed.

```
{
  "encryptedChoice":{
    "ciphertexts":[
      {"x":"0296EA334615B205F2B75AED751586FBFBFF794B4F96780146E55A11D3ED5447BF",
       "y":"0237A9A3B7738311C6F36D954A8CAB89A697FD8AEF38676D732EC44FB978269F26"},
```

```
      {"x":"029701753C446CCAF47A37D6AC28107AB026DD914D77989D36CF0F9319D161297F",
       "y":"03793ED5EE4A3CD89BD74C4AE44E88614845B72702FCA623F54EEDE5821F7F453C"}]},
  "proofOfKnowledgeOfEncryptionCoins":[
    {"c":"5566761242412747901695976811575830955448754520688763805956328758729826961718 0",
     "f":"767504419574287542733664580636234298217746465290738331953900733675929417238 01"},
    {"c":"8749719116114204360635525281063307451869531242888872928555639793490860941811 9",
     "f":"1008000682756793106633911491383684373472672019661210123119370233554573875456 11"}
  ],
  "proofOfKnowledgeOfPrivateCredential":{
    "c":"712962940667273900171425732724991103536513324753112280445722255701621221994 58",
    "f":"937407930704449658343507317008112882918978770200390842342690696667654228524 27"
  }
}
```


Ballot revocations. We now describe how to process revocation tokens and the corresponding authorisation in order to determine which ballots should be flagged as revoked.

The election supports ballot revocations only if a revocation policy is included in the registry record. Otherwise; the ballot revocations are not supported and all correct ballots should take part in the tallying (the revocation and revocation authorisations boards are expected in this case to remain empty, but even if for some reasons there are not empty, their content should be ignored).

Note. *A revocation policy is represented by a record of type RevocationPolicy (Section E.37) optionally included in the registry record published on board registry (Section D.1.2).*

A revocation policy specifies the minimal required number m of election administrators (the so-called *threshold*) who need to authorise a revocation token in order for such a token to become valid. It also includes a list public PGP keys of the administrators. The listed public keys are of type `String` and contain a Base64 encoding of the content of the `.asc` file containing the public key of the election administrator (such a file can be obtained by exporting the *public key only* from the PGP/GPG Keychain application).

In the special case where the threshold m is 0, all the published revocation tokens are considered valid by definition (no further signed token authorisation are necessary). In this case, the Election Administrator Panel takes responsibility of correct handling of revocation tokens, i.e. it makes sure to only publish revocation tokens which are acknowledged by election administrators accordingly to the configured policy. (Note that even in this simple case the revocation process is fully transparrent: all the revocation tokens are published on the dedicated bulletin board and visible to the auditors.) In this case, the list of PGP keys can be empty, and a verification tool does not need to implement PGP signature checking described below.

Note. *Unauthorised revocation tokens are published on board revocations (Section D.1.5). They are represented as records of type RevocationToken (Section E.38). Authorisations are published on board revocation-authorisations (Section D.1.4) and are represented as records of type RevocationTokenAuthorisation (Section E.39).*

A revocation token contains an election identifier and a list of voter identifiers. The election identifier in a revocation token should be the same as the election identifier in the registry record; if it is not the case, the revocation token is malformed and should be ignored.

The *normalized representation* of the token (relevant for PGP signature checking process, not relevant if $m = 0$) is defined as the string of the form

$$\texttt{REVOCATION\_TOKEN\{ELECTION=}\mathit{electionId}\texttt{,VOTERS=[}\mathit{voterId}_1\texttt{, ... ,}\mathit{voterId}_n\texttt{]\}}$$

For instance, a token with election identifier `XA78` and voters `voter501` and `voter809` has the normalized representation

```
REVOCATION_TOKEN{ELECTION=XA78,VOTERS=[voter501,voter809]}
```

The *fingerprint* of a token is generated by computing the SHA-256 hash function of the normalized representation of the token and representing the first 10 bytes of this hash as a hexadecimal string.

For instance, the fingerprint of the token from the example above is `1f515cc47433d46a89be`.

A revocation token is authorised, if there is at least m valid authorisations published on the revocation authorisations board, issued by m *distinct* election administrators.

The procedure of checking an authorisation is as follows. An authorisation contains:

- The public key of the election administrator which issued this authorisation, in the format explained above.
  This should be one of the public keys listed in the revocation policy. If this is not the case, the authorisation is invalid.
- The fingerprint of the token which is authorised.
  A revocation token with this fingerprint should be published on the revocation board. If this is not the case (which is not expected), this authorisation should be ignored.
- The PGP signature of the revocation token (with the given fingerprint) issued by the holder of the secret key corresponding to the included public key. If this signature is incorrect, the authorisation is not valid.
  More specifically, the signature is on the normalized representation of the token. The signature is of type `String` and contains a Base64-encoded content of a *detached* PGP signature file, as can be obtained, for instance, by command of the form:

  ```
  gpg –default-key USER –output SIGNATURE.sig –detach-sig INPUTFILE
  ```

  where `INPUTFILE` contains the normalized representation of the token.

If all the above checks succeed (the public key in the authorisation is valid, the fingerprint refers to one of the published revocation tokens, and the signature is correct), the authorisation is valid.

A.6 Verifiable Shuffle

In this section we describe the way in which ballots are shuffled and how this shuffle can be verified. We use construction proposed by Wikströmet al. [9, 10] which is used, amongst others, in Verificatum [11]. Specifically we take the optimised variant for ElGamal as it is presented in Haenni et al pseudo-code algorithms for implementing Wikström's verifiable mix net [5], while extend the construction to support parallel shuffles (where one can shuffle *multi-ciphertexts*, that is ciphertext consisting of a number of simple ElGamal ciphertexts). For completeness, we provide the proofs in our technical report [6].

Note. *The packets to be shuffled are read from board mixing-input-packets (Section D.7.1). The shuffled packets are published on board mixing-mix-Polyas (Section D.7.2). The verification procedure should check that the output mix packets contain valid zero-knowledge proofs, as described below, with respect to the corresponding input packets. .*

A verifiable shuffle in the simple case (that is without parallel shuffles), takes a list of ciphertexts, which it re-encrypts and shuffles to produce an output list of ciphertexts. More specifically, a cryptographic shuffle of ElGamal encryptions $\mathbf{e} = (e_1, ..., e_N)$ is another list of ElGamal encryptions $\mathbf{e'} = (e'_1, ..., e'_N)$, which contains the same plaintexts in permuted order. Additionally, a party that carries out shuffling produces an evidence to prove that input and output ciphertext are in fact in the described relation. This called a *proof of shuffle*. In the extended case (that we consider in this section) e_i, e'_i are multi-ciphertexts (sequences of ElGamal ciphertexts of some fixed length).

The algorithm assumes the following **setup**:

- a cyclic group G of prime order q in which both the decisional and computational Diffie-Hellmen problems are hard,
- the public key pk,
- a multi-commitment key ck, containing independent generators $h, h_1, ..., h_N \in G$.

Selecting independend generators. It is critical, that the elements of the multi-commitment key are indeed generated independently (security of extended Pedersen commitments depends on the assumption that one does not know/cannot compute the discrete logarithm of one element of the commitment key in the basis of another one). To this end, we generate the multi-commitment key in the reproducible and pseudo-random way, as follows:

$$\begin{aligned} h &= gen(10) \\ h_i &= gen(10 + i) \qquad \text{for } i = 1, \ldots N, \end{aligned}$$

where $gen(j)$ is the generator obtained using Algorithm 7 with parameter *index* set to j and parameter '*seed*' set to (the byte representation of the string) `"Polyas"`. Note that Algorithm 7 works for elliptic curve based groups of prime order, such as the group `secp256k1` we use in our system instance; for different groups, analogous methods should be used.

Example. For the used secp256k1 group, the first few elements of the multi-commitment key, generated as above, are (represented in the compressed form of ANSI X9.62 4.3.6):

h = 02549196EA21197151C73C3C9BDA1F12DA2BBEA99F2EFB0DD8BC235A9CED37ECB9

h_1 = 03F08FCB284F32B737E0529840334D481E055AD6AFA18AB91A3B02939EB19EB8DD

h_2 = 034A90A88BD2D3A92D7A29D19135F25536516D46FE4B8776C74B9E26D834FDA588

h_3 = 0365DB947FD33BE257599D9E0BD1513E6F7B3BBE6C9008382E22F4B527D3A39299

h_4 = 031E9073F6821FADE4307507F0D2756EFAAA4522FC15391390C943F4F4D9F32CE5

The **interactive zero knowledge proof** of the verifiable shuffle, run by the prover $\mathcal{P}$ and the verifier $\mathcal{V}$, is given in Algorithm 10, where we use the following notation.

- We use the multiplicative notation for the group operation. As usually, by $\mathbb{Z}_q$ we denote the field of integers modulo the prime q.
- A^N denotes the set of vectors of length N containing elements of A. We will typically denote vectors in bold, for instance $\mathbf{a}$. We will denote the i-th element of such a vector using subscript, for instance as $\mathbf{a}_i$.
- Similarly, $A^{N\times N}$ is the set of square matrices of order N containing elements of A. We will denote matrices using upper case letters, for instance M. By $M_{i,j}$ we denote the element of M in the i-th column and j-th row.
- A matrix M is a *permutation matrix*, if it contains only 0 and 1 values and, moreover, if every column and every row contains exactly one 1.
- $M\mathbf{x}$, for $M \in \mathbb{Z}_q^{N\times N}$ and $\mathbf{x} \in \mathbb{Z}_q^N$, is a vector of length N where i-th position is equal to $\sum_{j=1}^{N} M_{j,i}\,\mathbf{x}_j$.
- For a permutation π of the set $\{1,\ldots,N\}$, we will denote by M_π the *permutation matrix defined by* π which is the permutation matrix such that, for all $i \in \{1,\ldots,N\}$, value 1 in the i-th column is at position $\pi(i)$. It follows that, if $\mathbf{y} = M_\pi\mathbf{x}$, then we have $\mathbf{x} = (\mathbf{y}_{\pi(1)},\ldots,\mathbf{y}_{\pi(N)})$.
- $EPC_{h,h_1,\ldots,h_N}(\mathbf{m}, r)$, for $\mathbf{m} \in \mathbb{Z}_q^N$ and $r \in \mathbb{Z}_q$, is defined as $h^r \prod_{i=1}^{N} h_i^{\mathbf{m}_i}$ (otherwise known as an extended Pedersen commitment).
- $Com(M, \mathbf{r})$, for $M \in \mathbb{Z}_q^{N\times N}$ and $\mathbf{r} \in \mathbb{Z}_q^N$, is $(\mathbf{c}_1, ..., \mathbf{c}_n)$ where $\mathbf{c}_i = h^{\mathbf{r}_i} \prod_{j=1}^{N} h_j^{M_{i,j}}$, which means that $\mathbf{c}_i$ is the extended Pedersen commitment to the i-th column of M.
- A *ciphertext* is a pair $(x, y) \in G_q$. By $(x, y)^z$ we denote (x^z, y^z). Ciphertexts can be multiplied component-wise, that is $(x, y)(x', y') = (xx', yy')$. One can also compute component-wise the z-th power of a ciphertext, that is $(x, y)^z = (x^z, y^z)$.
- $\mathbf{ReEnc}_{g,pk}(e, r)$, for $e = (x, y) \in G^2$ and $r \in \mathbb{Z}_q$ is $(xg^r, y\,pk^r)$.
- $\mathbf{ReEnc}_{g,pk}(\mathbf{e}, \mathbf{r})$, for $\mathbf{e} \in (G^2)^w$ and $\mathbf{r} \in \mathbb{Z}_q^w$, denotes the sequence $\mathbf{ReEnc}_{g,pk}(\mathbf{e}_1, \mathbf{r}_1), \ldots, \mathbf{ReEnc}_{g,pk}(\mathbf{e}_w, \mathbf{r}_w)$.

Algorithm 10: Interactive ZK-Proof of Extended Shuffle

Common Input : group generator $g \in G$, public key $pk \in G$,
multi-commitment key $h, h_1, ..., h_N \in G$,
matrix commitment $\mathbf{c} \in G^N$,
ciphertext vectors $\mathbf{e}_1, ..., \mathbf{e}_N \in (G^2)^w$ and $\mathbf{e}'_1, ..., \mathbf{e}'_N \in (G^2)^w$.

Private Input of $\mathcal{P}$: Permutation π of the set $\{1, \ldots, N\}$, randomness $\mathbf{r} \in \mathbb{Z}_q^N$ and randomness $R \in \mathbb{Z}_q^{N \times w}$, such that $\mathbf{c} = Com(M_\pi, \mathbf{r})$ and $\mathbf{e}'_i = \mathbf{ReEnc}_{pk}(\mathbf{e}_{\pi(i)}, R_{\pi(i)})$.

1 $\mathcal{V}$ chooses $\mathbf{u} \in \mathbb{Z}_q^N$ randomly and sends it to $\mathcal{P}$.

2 $\mathcal{P}$ computes $\mathbf{u}' = M\mathbf{u}$. Then $\mathcal{P}$ chooses $\hat{\mathbf{r}} \in \mathbb{Z}_q^N$ at random and computes

$$\bar{r} = \mathbf{r}_1 + \cdots + \mathbf{r}_N, \qquad \tilde{r} = \langle \mathbf{r}, \mathbf{u} \rangle,$$
$$r^\diamond = \hat{\mathbf{r}}_N + \sum_{i=1}^{N-1} \left(\hat{\mathbf{r}}_i \prod_{j=i+1}^{N} \mathbf{u}'_j \right), \qquad \mathbf{r}^\star = R\mathbf{u}$$

$\mathcal{P}$ randomly chooses $\hat{\boldsymbol{\omega}}, \boldsymbol{\omega'} \in \mathbb{Z}_q^N$, $\omega_1, \omega_2, \omega_3 \in \mathbb{Z}_q$, and $\boldsymbol{\omega_4} \in \mathbb{Z}_q^w$, and sends the following values to $\mathcal{V}$:

$$\hat{c}_0 = h_1, \;\; \hat{c}_i = h^{\hat{\mathbf{r}}_i} \hat{c}_{i-1}^{\mathbf{u}'_i} \;\; (i \in \{1, \ldots, N\}) \qquad t_1 = h^{\omega_1} \qquad t_2 = h^{\omega_2} \qquad t_3 = h^{\omega_3} \prod_{i=1}^{N} h_i^{\boldsymbol{\omega}'_i}$$

$$\mathbf{t}_4 = \mathbf{ReEnc}_{g,pk}(\prod_{i=1}^{N} \mathbf{e}'^{\boldsymbol{\omega}'_i}_i, -\boldsymbol{\omega}_4) \qquad \hat{\mathbf{t}}_i = h^{\hat{\boldsymbol{\omega}}_i} \hat{c}_{i-1}^{\boldsymbol{\omega}'_i} \;\; (i \in \{1, \ldots, N\})$$

3 $\mathcal{V}$ chooses a challenge $c \in \mathbb{Z}_q$ at random and sends it to $\mathcal{P}$.

4 $\mathcal{P}$ then responds with

$$s_1 = \omega_1 + c \cdot \bar{r} \qquad s_2 = \omega_2 + c \cdot r^\diamond \qquad s_3 = \omega_3 + c \cdot \tilde{r} \qquad \mathbf{s}_4 = \boldsymbol{\omega}_4 + c \cdot \mathbf{r}^\star$$

$$\hat{\mathbf{s}} = \hat{\boldsymbol{\omega}} + c \cdot \hat{\mathbf{r}} \qquad \mathbf{s}' = \boldsymbol{\omega'} + c \cdot \mathbf{u}'$$

5 $\mathcal{V}$ accepts if and only if

$$t_1 = (\prod_{i=1}^{N} \mathbf{c}_i / \prod_{i=1}^{N} h_i)^{-c} h^{s_1} \qquad t_2 = (\hat{c}_N / h_1^{\prod_{i=1}^{N} \mathbf{u}_i})^{-c} h^{s_2} \qquad t_3 = (\prod_{i=1}^{N} \mathbf{c}_i^{\mathbf{u}_i})^{-c} h^{s_3} \prod_{i=1}^{N} h_i^{\mathbf{s}'_i}$$

$$\mathbf{t}_4 = \mathbf{ReEnc}_{g,pk}((\prod_{i=1}^{N} \mathbf{e}_i^{\mathbf{u}_i})^{-c} \prod_{i=1}^{N} \mathbf{e}'^{\mathbf{s}'_i}_i, -\mathbf{s}_4) \qquad \hat{\mathbf{t}}_i = \hat{c}_i^{-c} h^{\hat{\mathbf{s}}_i} \hat{c}_{i-1}^{\mathbf{s}'_i}$$

- $\langle \mathbf{a}, \mathbf{b} \rangle$, for $\mathbf{a} \in \mathbb{Z}_q^N$ and $\mathbf{b} \in \mathbb{Z}_q^N$ is $\sum \mathbf{a}_i \mathbf{b}_i \mod q$.
- $a\mathbf{x}$, for $a \in \mathbb{Z}_q$ and $\mathbf{x} \in \mathbb{Z}_q^N$, is a vector of length N where i-th position is equal to $a\mathbf{x}_i$.
- For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_q^N$ we write $\mathbf{x} + \mathbf{y}$, to denote the pairwise addition.

Note that, in Step 5, $\mathbf{e}_i$ is a multi-ciphertext (that is an element of $(G^2)^w$) and $\mathbf{u}_i$ is a number in $\mathbb{Z}_q$; the exponentiation is, therefore, applied component-wise.

This interactive algorithm is in fact a correct zero-knowledge protocol. For completeness, we provide the precise security statement as well as the proof in our technical report [6].

We will now present the non-interactive version of this zero-knowledge proof which is used in our e-voting system. This non-interactive version is derived from Algorithm 10 in the standard way using the Fiat-Shamir heuristic.

Algorithm 11: ZK-Proof of Shuffle Generation

Public Setup:

- group G of order q with a generator $g \in G$, $pk \in G$,
- multi-commitment key $h, h_1, \ldots, h_N \in G$

Input :

- lists of input multi-ciphertexts $\mathbf{e}_1, ..., \mathbf{e}_N$ and output multi-ciphertexts $\mathbf{e}'_1, ..., \mathbf{e}'_N$, where $\mathbf{e}_i, \mathbf{e}'_i \in (G^2)^w$,
- permutation π of the set $\{1, \ldots, N\}$,
- random coins $\mathbf{r} \in \mathbb{Z}_q^N$ and $R \in \mathbb{Z}_q^{N \times w}$ (as in Algorithm 10),

1 Derive a challenge $\mathbf{u}_i \in \mathbb{Z}_q$ (for $i \in \{1, \ldots, N\}$) from the following values (in the given order):

$$g,\ pk,\ h,\ h_1, \ldots, h_N,\ \mathbf{e}_1, \ldots, \mathbf{e}_N,\ \mathbf{e}'_1, \ldots, \mathbf{e}'_N,\ \mathbf{c},\ i$$

using Algorithm 4 (uniform hash) with parameter 'q' set to q (the group order), where $\mathbf{c} \in G^N$ is the commitment to the permutation matrix with randomness $\mathbf{r}$, that is $\mathbf{c} = \mathit{Com}(M_\pi, \mathbf{r})$

2 Carry out Step 2 of Algorithm 10, defining $\hat{c}$ as $\hat{c}_1, \ldots, \hat{c}_n$ (notice that $\hat{c}$ does not include $\hat{c}_0$ which is simply defined as h_1).

3 Derive a challenge c from the following values (in the given order):

$$g,\ pk,\ h,\ h_1, \ldots, h_N,\ \mathbf{e}_1, \ldots, \mathbf{e}_N,\ \mathbf{e}'_1, \ldots, \mathbf{e}'_N,\ \mathbf{c},\ \hat{c},\ t_1,\ t_2,\ t_3,\ \mathbf{t}_4, \hat{\mathbf{t}}$$

using Algorithm 4 (uniform hash) with parameter 'q' set to q (the group order).

4 Carry out Step 4 of Algorithm 10.

5 Output the ZK-proof consisting of $(\mathbf{c}, \hat{c}, t_1, t_2, t_3, \mathbf{t}_4, \hat{\mathbf{t}}, s_1, s_2, s_3, \mathbf{s}_4, \hat{\mathbf{s}}, \mathbf{s}')$.

The non-interactive procedure for zero-knowlege proof generation is given in Algorithm 11.

As one can see, this procedure follows the steps of the prover from Algorithm 10, where the challenges are derived from the proof statement and the protocol transcript (up to the given point) using a hash function.

The corresponding non-interactive procedure for zero-knowlege proof verification is given in Algorithm 12. This algorithm derives the challenges in the same way as it is done in Algorithm 11 and checks the equation of the last step of Algorithm 10.

Algorithm 12: Verification of a ZK-proof of Shuffle

Public Setup :

- group G of order q with a generator $g \in G$, $pk \in G$,
- multi-commitment key $h, h_1, \ldots, h_N \in G$

Input :

- lists of input multi-ciphertexts $\mathbf{e}_1, ..., \mathbf{e}_N$ and ouput multi-cipherexts $\mathbf{e}'_1, ..., \mathbf{e}'_N$, where $\mathbf{e}_i, \mathbf{e}'_i \in (G^2)^w$,
- the ZK-proof $(\mathbf{c}, \hat{c}, t_1, t_2, t_3, \mathbf{t}_4, \hat{\mathbf{t}}, s_1, s_2, s_3, \mathbf{s}_4, \hat{\mathbf{s}}, \mathbf{s}')$, where

$$\mathbf{c} \in G^N, \quad \hat{c} = (\hat{c}_1, \ldots \hat{c}_n) \in G^N,$$
$$t_1 \in G, \quad t_2 \in G, \quad t_3 \in G, \quad \mathbf{t}_4 \in (G^2)^w, \quad \hat{\mathbf{t}} \in G^N,$$
$$s_1 \in \mathbb{Z}_q, \quad s_2 \in \mathbb{Z}_q, \quad s_3 \in \mathbb{Z}_q, \quad \mathbf{s}_4 \in \mathbb{Z}_q^w, \quad \hat{\mathbf{s}} \in \mathbb{Z}_q^N, \quad \mathbf{s}' \in \mathbb{Z}_q^N,$$

Note that $\hat{c}$ does not include $\hat{c}_0$ which is defined as h_1.

1 Derive $\mathbf{u}_i \in \mathbb{Z}_q$ (for $i \in \{1, \ldots, N\}$) from the following values (in the given order):

$$g,\ pk,\ h,\ h_1, \ldots, h_N,\ \mathbf{e}_1, \ldots, \mathbf{e}_N,\ \mathbf{e}'_1, \ldots, \mathbf{e}'_N,\ \mathbf{c},\ i$$

using Algorithm 4 (uniform hash) with parameter 'q' set to q (the group order).

2 Derive a challenge c from the following values (in the given order):

$$g,\ pk,\ h,\ h_1, \ldots, h_N,\ \mathbf{e}_1, \ldots, \mathbf{e}_N,\ \mathbf{e}'_1, \ldots, \mathbf{e}'_N,\ \mathbf{c},\ \hat{c},\ t_1,\ t_2,\ t_3,\ \mathbf{t}_4, \hat{\mathbf{t}}$$

using Algorithm 4 (uniform hash) with parameter 'q' set to q (the group order).

3 Carry out Step 5 of Algorithm 10. Output 'true' if and only if all the equations of this step hold true. Notice that $\hat{c}$ (given as the input to this algorithm) does not include $\hat{c}_0$ (a value necessary to compute $\hat{\mathbf{t}}$) which is simply defined as h_1.

We emphasise that, for all the shuffle proofs generated for an election, the fixed, verifiably generated multi-commitment key is used, as specified earlier in this section. Therefore, for shuffle proofs generated for election instances specified in this document, **the verification algorithm should use this fixed multi-commitment key**.

Example: We now present an instance of a valid proof of shuffle for the group `secp256k1` and the standard fixed multi-commitment key (as defined earlier in this section). The used

public key is:

$$pk = \texttt{030000000000000000000000003B78CE563F89A0ED9414F5AA28AD0D96D6795F9C63}$$

The sequence $\mathbf{e}_1, ..., \mathbf{e}_N$ of input multi-ciphertexts with $N = 5$ and $w = 3$ is given in the JSON format:


```
[ {
  "ciphertexts" : [ {
    "x" : "0214C8ED687A03590A5DFF16207636B75641E4265077C0C8546FF820188662EB5F",
    "y" : "03E08BA84715A5CC14B27895F7A42194933CF7D9B71101F99CDEB0800399F02C20"
  }, {
    "x" : "032392DF7D3B7BDFFF9A7ACC599E20D4C435755B4EF73C458900B0928B34864DAE",
    "y" : "02234DBE74A1D90B6D7EE8A376958347028B07A71DEB69FD203692AA9CCE86865A"
  }, {
    "x" : "02B4EA4738B32421826F1F87A4712372059F4DEDBA136155BAAEA76FDA77FD1B77",
    "y" : "03577F6FEFC44A662ACA61A09A3C0D6D57B5BBAE5D54DEC8843F1A3CBC5096B557"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "0203B5E69A3B368C16351D2AFB94135A13C3E3E325F0D04612F31FC8E79CA8AF23",
    "y" : "020833D683DC989F9F43CC5C6229A01D8036E4B0EF980E9B3A5035D4267E85B328"
  }, {
    "x" : "02305032260DE32A5C1C2E38AB18229AD1A52540A1333E265EEDA3060BC84EDD10",
    "y" : "024FC453BDF7159175F8A647DDD81E124A9BC988AFDEAE4F578A0A0F1C7AFD01C4"
  }, {
    "x" : "029EE3273853C61A09EE816D93220D3BB268E843B84516DB3E19108FEC719E9B29",
    "y" : "036AB3656C933BF8A6C07A1031FBB502250CB966A25BA1A0E668718D20DCC594CC"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "020D5725F57B7B90DC5D16A45C121F8B8ACF2981490E5BF7756F78E1FFC9010364",
    "y" : "0338E08940BC3F3681DB354930ACEB98C4EBB7C7796225DD274D5E9B828A18FDC8"
  }, {
    "x" : "03AE8B4B0DC1A0DCE73DC5BE8F8F45B76D349FA77A22109D0647890B035DF1C18D",
    "y" : "025718470F1BE2A75653CD608CF2999E5A1F7B57778C76DB5B74AED2084C87B624"
  }, {
    "x" : "02811B7C71E2C5EC448FB9E937FB97501E2DB95B080C0FC3639EC25FBE06BB859F",
    "y" : "038CCFC46E348B0A2D5FF4DB1EFA4E4388D706D133B13B35A1D0D582B50A2B4495"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "02DC464302FBE1CE58ECE95B1A40B4FD99C190456844E17D62AA51801C246668F2",
    "y" : "033AC4F52C385066CBBBC04BDC87D87232F58B02B98F7F4661C766114FDD950A65"
  }, {
    "x" : "032C50FD2FB9811BC47A76D1478E13A3E5B4631AF8B97D3EB32347FB597FC0D72C",
    "y" : "02FFA0CFD2B1E7CA591BFA8E33AB9EB54005A773DF5E0A96371DF3ADCDDC58D965"
  }, {
    "x" : "023460FDA0AF0D22DFF6F18C2CDD7A8325EB9EA28C177D83CB2BB2CD7FE10ADE66",
    "y" : "03ED73B8D97CEED2B56924D91A1C48330DA0669CA2323463C4EC211429704E2C2D"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "024622D7BD63EA32760775DEA4F70531445C22BBFD4E4BCDB9B65E93E728916DF1",
    "y" : "03D21229F0D579ACBDCBA52A69D452FFA15BE431AB44E053F8C045690643AFEEA4"
  }, {
    "x" : "0292B9CE0CEA817BDA811CD1B0687F18EEA410FF4AED585E57F2CB2772AEB12089",
    "y" : "03215C6F22892AF7AFF3D07D5945C33E0ECA2A71114E7735182E0A426925A3B943"
```

```
  }, {
    "x" : "039B21A650670E8C9266A8B9951CA667C417ABF10D872FC518295CEF251365CC51",
    "y" : "0341B2AE82C303AA11EB36F888C0EE84A3CB45EB3CC2EA2A9892453876E1A63732"
  } ]
} ]
```

The sequence $\mathbf{e}'_1, ..., \mathbf{e}'_N$ of output multi-ciphertexts is given in the JSON format:

```
[ {
  "ciphertexts" : [ {
    "x" : "02456DE1179DB1B71A99277A3DE50F4AABB8F067E919394094F1796975D61C8241",
    "y" : "0204B89C49C36D914A6345FC891B2D2C218CD6EC92618246D6CBCEDC8B3915E7B3"
  }, {
    "x" : "0209C4E8F415C4BD201783206EACBF2350D621DDB54AFBBE530FA4FA01618DF539",
    "y" : "02BA2F9D6BA01F64B36DE7BF8C0B6FFB54C3E4E8E3A208353827F415701C07E8F8"
  }, {
    "x" : "0369715046EA5CA8A5BB300ADDC7FA7DC46A7F3CDBB5F0E3F8DB91D018B8C9C973",
    "y" : "031ED897C05609D58C216BA04B7C59E914CCA2CF260F18E9C9E61831DEE9FBB47F"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "020C1C7A58906A5E7FEA11D8499F3FC6807F8A9214E82D0448E7D1128BC7BA216F",
    "y" : "03BC78094A4296878A77FE026D689FD10FD5DE3005FF512BA19E599AB9EE51D46B"
  }, {
    "x" : "036CD0CC7159AE847A80BFCC26088F919EA808343CC631E6B7D17534AB9364A1C2",
    "y" : "036721AB124B5E691355E269ED85EC7225DF2F1E2689A3904E63CDB99557DE590E"
  }, {
    "x" : "02813373C27103E17FD43EBFBEF79CA7BA31F3F17E707B42362B7848063D10A271",
    "y" : "032C9A98CDEB02444DF56B09553C36FCBDB9B470B1A39518EAFD64018F35999262"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "03AB7BE97B4A7B879B9F893C03AA7B477BF34785F80FC967EC75144882BB8EA72C",
    "y" : "02A641223FC94FEE1E0E7F24E1C1B475BA45F91364984D94F3FB529812C26EBC34"
  }, {
    "x" : "02F7340EAA4299973AB27231C27E53B90B913554F03721BA06E3C344DE72F42A0C",
    "y" : "02128077E42835DB0B2A992BA9A3415481C2EC1A9350C07674111CFFD3528348C6"
  }, {
    "x" : "0372B6FAD5684760BA7105A449112D78EF4F60CC80752B33CCB79852AFF36BEAC7",
    "y" : "02A284A3A53AE803E684F5E903DB687354F7C21CFA51C304691C1D5DE0394883F9"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "0279045EEC54A7C574B0475932ACA3F7C5022DB5DA63CCF59486037E87FFF5FC13",
    "y" : "0339C7B1DAFF747CA0BB51199909C51EE1A2A8D85678CFFCC10834CA850F923BA5"
  }, {
    "x" : "0287DBE4AC3C675718332715FD10B8A65F75A79769F664B8C9EF7138BE27DCEC3D",
    "y" : "022F8D43DEA945BC03299F4285CDE8FF028EAA7352826490F1DE2C8A3E7200541F"
  }, {
    "x" : "03A06D4592A73B1211C0E7616C502CC1A851CE869F6474A24869B5D692D022BE83",
    "y" : "03F43A46AC5B017F8C090B1BB19FFBF91FC0BBD635980BE36028A07C9A78AE73CC"
  } ]
}, {
  "ciphertexts" : [ {
    "x" : "033BCD27079C93D3A0D5489EA4E97A37AD2C0BB993AB03828F15AFDCE86A09D1EE",
    "y" : "02E20CC62C67B3D7ACA45D35F0982ADB2AFE00B211491E2BEE57FF839E2AA77891"
  }, {
    "x" : "0227403C521B76EF97C257042F95B67EF0EBCF38CBEF6147ED384A696D09B3B4F1",
    "y" : "02055EF3DF1BB93420BAC356B7E4FFF281F996E06955A67ED8E7A81D0F2E902030"
```

```
  }, {
    "x" : "03B4BF902DF1EB8BF992E1FCB7DAFBFA52DF056869879A4EFA2EE617224B2028AA",
    "y" : "02C5945ED05DD243D456DBD883BE62062CE160B7947C1931B16E9A742A643CA9A7"
  } ]
} ]
```

The proof of shuffle is given by the following JSON. Note that the components of t_4 are given in separate lists (first components in `t4x` and second components in `t4y`).

```
{
  "c" : [
    "03063CA66F8C0ECEAAB8236BA467F3D817710FBF45792A6AC31DF6AF4293F3F5FC",
    "038B87D18C31424A6C217FF70AA58F5B5682093DE6BCC7073E66DAE1BF6B32BE0D",
    "038C81504B353A3DA542AB71B2D9F303053305E675C2D6D09B59692C72C79BC6C7",
    "03EBE3705F74AF1E0E0E5CEF5C55A0C4768EF94B656799CF81A7D3A7664F01A1E7",
    "03CCAB079B02DFED7DB56621060FBF2A795E39D4CDEBA4A9B7799AB92191484DA5"
  ],
  "cHat" : [
    "02015DCE2FFFFCA395A6E62C0F2661BF4F4D17AEDBAD90B1CD3C111B7E9FEF0DCE",
    "03560966D35CF68F89F81E233D329F1C64E8F5991D270CEF49843BD00E6DC5CBF8",
    "027718D13F8E0AC62E269507BB3FBDACDDFF11D29E6971C64F2BFF9535735FF5CE",
    "033749E1628EEB14EE6AD370BD47F430020A3E7CD77A3D2CF20F5177EE79FDA462",
    "03EE78C0CD5E0320E8BC342C420D873802ED1FF2AE8D9D4FC165B405821394E82C"
  ],
  "t" : {
    "t1" : "035C091CBE9D77D0030FE3584CC20DF2C1DB0621F07E3C16600981C9A806C39E15",
    "t2" : "0211DA61A861A0B7AF9D3A77022EB55B511D4B07F18D271B6C61ADFF155BD419B1",
    "t3" : "023D82EA9E9ADC647E699EF1901BCBB89B7D587DAF328BA354B100020245C1E6E7",
    "t4y" : [
      "03919EDA2E6275ABA58260D4B0A260E8E6A2D9BB99486CFB1BAB41C56400FFF473",
      "039681898C617F1D9304B1B3103C009CDF98B6509A768C89B3DC1D6D97102414AD",
      "031E12721D6AB8E4789ABD2EB074BCBEE4B931112B23F45C5CBBA2040DF0CDBAFE"
    ],
    "t4x" : [
      "03C0CE5B3CB78C48072687ADB95A74928065C5E02197E8FA84CB82FFF5F32B56C2",
      "030DE4191A9F916DDA2BC0F89B6CF91FB86504A937116F0F5390F947CB19C9B2DF",
      "022204EADACC736A723A774F2F852DACE0FE2063A35B6DC73DB02CEE9C6913338E"
    ],
    "tHat" : [
      "02A6B5848F4C8A548DEC32CF69DD62102A0229A91754833011ABEDF7C37CAD0B35",
      "03AFFAFC3E79F3F0D5E167D28CD07E32FB2A2013E3C4E10228A1CF93EC01191D24",
      "02D4CC74684CAC1AAF3B097202A53D6C95E56063A9FAACFB83880A9275ECF1DB5F",
      "0247C0DAF54B4BBB57CD02055F669B6C3AF8A2772B7A4D724B028F80CC85F7905F",
      "02BD163CFCF9F8F5220D97C2CAEED5ED6F8631FA440EDFE62EF42A1130A10D60BF"
    ]
  },
  "s" : {
    "s1" : "691403168808870084292994334095347926968484190932168736805519748971448871596 10",
    "s2" : "5313118011581947160364966188583784895186789297060349189477824268230125480343 9",
    "s3" : "665458857868125362074344322226404779956310706434291603791212997678285299230 58",
    "s4" : [
      "492341498692505949593015148069278807616215946523829298649165364785672935764 03",
      "441193659166381032297790401462562314835381643071183629443588528578211165056 72",
      "864620110212562475105873567064380165955271689713539302546071869658332380876 71"
    ],
    "sHat" : [
      "875795230461220189878174010094612041211602460148606001189013588822592842856 35",
      "563146904573362360858578316798248944267389819046947251284595207672015103023 00",
```

```
      "257536798534188791908968909322706653316403580300576555504575090947276625951 42",
      "639151295138659111365976043406499088300393435819132114366249024296433583482 25",
      "698543450230464567523509154837940018459938656281809226591934970653287280763 15"
    ],
    "sPrime" : [
      "723662745720445281026699831910110329539914845880991217692466448825829088865 23",
      "173526105758966639214313353323165188051528846817692389351432880478731481083 74",
      "565956895154257803221671500599262699897573261875038883918507559583958350665 0",
      "932080007410437411838489719081959218615812557313705042807443409080219225173 10",
      "143571025506288364720115210080200571341712593730418929812999232067530764241 52"
    ]
  }
}
```


The values of the challenges $u_1, \ldots, u_5$ and c computed by the verification procedure (Step 1 and 2) are:

$$u_1 = 254231732614038380457804986591019291433742120248859617496771911238943454572 03$$

$$u_2 = 107163395173780552120959839682734915419294108435315434884329493449297702516 989$$

$$u_3 = 275272783996018798235989412651035600042038277764256857729806632192370523900 97$$

$$u_4 = 106382107453541024383519774022048119452768386444341851381719160092926187339 847$$

$$u_5 = 780254218556383017924390055501415336322183181230841877177947326431612393415 02$$

$$c = 148869579201420204254159707507132970444327099620757348033910292100254596992 80$$

A.7 Verifiable Decryption

The ballots, after being shuffled in a verifiable way (Section A.6), are decrypted by the Election Provider (Polyas). This decryption step is also verifiable, which means that appropriate zero-knowledge proofs are produced, allowing an independent party to make sure that this step has been carried out correctly, without revealing the used private key.

Note. *The decryption is applied to all the ciphertexts in all the output mix packets published on board mixing-mix-Polyas (Section D.7.2). For each such packet a corresponding packet with decrypted messages is published on board decryption-decrypt-Polyas (Section D.4.1), as explaned below.*

An encrypted ballot is represented as a *multi-ciphertext*, i.e. a sequence $e = e_1, \ldots e_n$ of simple ciphertexts, where every ciphertext e_i is of the form (x_i, y_i) with $x_i, y_i \in G$.

In the first step of the decryption process, for each $i \in \{1, \ldots, n\}$, a so-called decryption share $\alpha_i = x_i^{sk}$ is computed, where sk is the secret election key. From this value the decryption result $d_i \in [0, \mathit{messageUpperBound})$ can be easily computed: $d_i = \mathit{decode}(y_i/\alpha_i)$, where *decode* denotes the decoding function, which for a given group element returns an element in $[0, \mathit{messageUpperBound})$, as defined in Section A.2.1.

The values $d_1, \ldots, d_n$ are then combined and decoded into a binary message (a byte array) using Algorithm 6, where the parameter 'q' is set to *messageUpperBound*. This message is the result of the decryption.

For this result, the following *zero-knowledge proof of correct decryption* is constructed. For every $i \in \{1, \ldots, n\}$, a zero-knowledge proof π_i is produced which shows that the i-th decryption share α_i is valid. This proof is a non-interactive variant of the well-known zero-knowledge proof of knowledge of discrete logarithm equality [4], where we use the strong Fiat-Shamir heuristic [3]. This proof is of the form $\pi_i = (c_i, f_i)$, where c_i, f_i are generated by drawing a random number a_i from the set $\mathbb{Z}_q$ and computing

$$\begin{aligned} A &= g^{a_i} \\ B &= x_i^{a_i} \\ c_i &= \mathit{uniformHash}_q(g, x_i, pk, \alpha, A, B) \\ f_i &= a_i + c_i \cdot sk \mod q \end{aligned}$$

Note that, indeed, (c_i, f_i) is a non-interactive zero-knowledge proof of equality of discrete logarithms $\log_g(pk)$ and $\log_{x_i}(\alpha_i)$ (which both are equal to sk). The overall proof of correct decryption for the message above consists of the all the decryption shares and the corresponding proofs: $(\alpha_1, \pi_1, \ldots, \alpha_n, \pi_n)$.

Note. *The output of this step is published on board decryption-decrypt-Polyas (Section D.4.1), where the decrypted binary messages along with zero-knowledge proofs of correct decryption are organized into records of type MessageWithProof (Section E.27) and then aggregated in packets of type MessageWithProofPacket (Section E.28) and published on board decryption-decrypt-Polyas (Section D.4.1).*

In order to verify the correctness of the decryption operation one can use the procedure presented in Algorithm 13 (applying this algorithm to every decrypted message with a proof and the corresponding input multi-ciphertext e).

Example: In this example, we use, as before, the group `secp256k` as defined in Section A.2 and represent elliptic points in the hexadecimal representation of the compressed form of ANSI X9.62 4.3.6. For the public key

$$pk = \texttt{03403091F3E81EE0E125FC33614DBA1ADBA569A3F7C05F9B36587054151508D490}$$

and the multi-ciphertext

```
{
    "ciphertexts":[
        {"x" : "03B467D17D26AE0A29034C698A15F8E50C7DD17D43F2F088091479B8C0B9CFFC60",
         "y" : "037EA63D9BB3E1FE8AB74DB33E60DCA353878B9169D01585D53F59A30DB7029C16"}
    ]
}
```

the following record in JSON format is a valid decryption record for which the verification procedure should succeed.

```
{
    "message":"0106010202040100000000000000000",
    "proof":[
```

Algorithm 13: Verification of ballot decryption

Input:

- public parameters: parameters of the used group G (including the generator g and the group order q), and the public key $pk \in G$,
- a multi-ciphertext $e_1, \ldots, e_n$, where $e_i = (x_i, y_i) \in G^2$,
- decryption result *message* given as a binary message (byte array) and a zero-knowledge proof of correct decryption $(\alpha_1, \pi_1, \ldots, \alpha_n, \pi_n)$ with $\alpha_i \in G$ and $\pi_i = (c_i, f_i) \in \mathbb{Z}_q^2$.

1 Check that all the components of the input are in the expected domains.

2 **for** i *in* $1, \ldots, n$ **do**

3 Compute $A_i = \frac{g^{f_i}}{pk^{c_i}}$ and $B_i = \frac{{x_i}^{f_i}}{{\alpha_i}^{c_i}}$.

4 Compute $c'_i = \mathit{uniformHash}_q(g, x_i, pk, \alpha_i, A_i, B_i)$, where *uniformHash* denotes the uniform hash function defined in Algorithm 4.

5 Check that $c'_i = c_i$ and reject the record as incorrect if this is not the case (this check concludes verification the proof of discrete logarithm equality for bases g, x_i, and values pk, α_i).

6 Compute, for each $i \in \{1, \ldots, n\}$, $d_i = \mathit{decode}(y_i/\alpha_i)$, where *decode* denotes the decoding function which for a given group element returns an element in $[0, \mathit{messageUpperBound})$, as defined in Section A.2.1.

7 Combine the values $d_1, \ldots, d_n$ into a binary message using Algorithm 6, where the parameter 'q' is set to *messageUpperBound*, and check that it yields the message *message*; reject the record as incorrect if this is not the case (this check concludes that the decryption shares have been correctly combined into the final message).

```
        {
            "decryptionShare" : "0220BF3495CDBDFCE2D10EB330AB72A56FC67B7D12BDB9270BB18D896089787FC6",
            "eqlogZKP" : {
              "c" : "868559846572463420252616817490333044376876919350408254900162470579420469772 71",
              "f" : "977490878123748274575683183135506792776051948271702216992646207926576238306 05"
            }
        }
    ]
}
```


A.8 Final Ballot Tallying

In this section we describe how decrypted ballots in a binary format are decoded and interpreted and then how the final result is computed.

An encoded ballot may, in the general case, contain encoded voter's choices for more than one ballot sheet (the voter may be eligible to vote for one or more ballot sheets). The list of these ballot sheets is specified by the *public label* associated with the ballot (as we will see below, encoded ballots are grouped in packets sharing the same public label and, hence, the same list of ballot sheets). More precisely, a public label is a concatentation of ballot sheet identifiers, where the `":"` character is used as a separator. For instance, public label `"a:b:c3"` represents the list of ballot identifiers `a`, `b`, `c3` (the order is significant); therefore an encoded ballot with such a label is supposed to encode voter's choices for these three ballot sheets.

Note. *To decode an encoded ballot, one needs to know the exact structure the ballot sheets (with the identifiers specified by the public label). The Structure of ballot sheets is published on board registry (Section D.1.2), as a record of type Registry (Section E.36). Such a record contains, under the field* `ballotStructures`*, a list of records of type Core3Ballot (Section E.14), each of which sperifies one ballot sheet. A record of this type contains, amongst the others, the field* `id` *with the ballot identifier, referred to by a public label.*

Note. *Decrypted ballots in binary format (represented as hexadecimal strings are published on board decryption-decrypt-Polyas (Section D.4.1), where decrypted messages (along with additional proofs of correct decryption) are grouped in records of type MessageWithProofPacket (Section E.28). Every such a record includes a public label (which specifies the list of ballot sheet identifiers, where evey ballot included in this recored encodes choices for these ballot sheets) and a list of records of type MessageWithProof (Section E.27), where field 'message' contains the decrypted ballot. All these decrypted ballots, together with the public label, constitute the input to the final counting.*

Structure of a ballot sheet. The structure of a ballot sheet, defined by a record of type Core3Ballot (Section E.14), is the following. A ballot sheet contains, in particular, a list of records of type CandidateList (Section E.8). Each such a record contains a list of records of type CandidateSpec (Section E.9) representing candidates (voting options). These records describe how a ballot should be presented to the voter (title of the ballot,

texts to be displayed, number of check boxes next to different voting options, etc.), as well as validation rules, for instance, the minimum and maximum number of votes a voter can select for a given candidate or a given candidate list. Note that if an optional field (such as ‘CandidateList.maxVotesTotal‘) is not present (or defined as ‘null‘) the corresponding constraint does not apply.

A voter can mark the whole ballot sheet as invalid (if this is allowed by the ballot specification; see field `showInvalidOption`). A voter can select chosen candidates/voting options (from one or more candidate lists). In the case of list voting, a voter can also give her vote for a whole list. The ballot sheet specification specifies how many options can be selected and if list voting is allowed; see ballot validation below.

Ballot encoding and decoding. To encode voter's choice as a byte array, the following simple method is used. Starting with the empty byte array B, we first append to B one byte indicating if the voter marked the ballot as invalid (1) or not (0). Then we iterate over consecutive candidate lists (the order is significant), and for each candidate list:

- we append to B one byte containing the number of votes for the whole candidate lists (0 if list voting is not allowed)
- for every consecutive candidate on this candidate lists, we append to B one byte containing the number of votes for this candidate.

If a voter can vote for more than one ballot sheet, we simply concatenate the encodings for consecutive ballot sheets (again, the order is significant).

The decoding procedure recovers the number of votes for different voting options (whole candidate lists and individual candidates), as well as the flags indicating which ballots are marked as invalid in a straightforward way.

Example. Consider a voter who can vote for two ballot sheets A and B, where ballot sheet A contains two candidate lists, each with two candidates, while ballot sheet B contains only one candidate sheet with three candidates. Then, the sequence of bytes

$$0, 0, 2, 3, 0, 4, 5, 1, 0, 7, 7, 7$$

encodes the following voter's choice:

- The choice for the first ballot sheet (ballot sheet A) is encoded by the bytes 0, 0, 2, 3, 0, 4, 5 and means that
  - the ballot is not marked as invalid (first 0),
  - the voter gives 0 votes for the first candidate list and the candidates on this list get, respectively, 2 and 3 votes,
  - the voter gives 0 votes for the second candidate list and the candidates on this list get, respectively, 4 and 5 votes.

- The choice for the second ballot sheet (ballot sheet *B*) is encoded by the bytes 1, 0, 7, 7, 7 and means that
  - the ballot is marked as invalid (first 1),
  - the voter gives 0 votes for the first (and the only) candidate list and the candidates on this list get 7 votes each.

A binary encoded voter's choice which cannot be decoded (is too long short/too long, has wrong value of the 'mark as invalid' field) is discarded as malformed. In such a case the whole voter's encoded ballot is discarded (we do not try to decode ballots partially).

Validation rules. Once a ballot is successfully decoded, each decoded ballot sheet needs to be validated against the rules expressed in the ballot specification. Consequently, only valid ballot sheets from well-formed ballots contribute to the final result. Note that, if a ballot sheet is invalid, we discard only this ballot sheet (not the whole ballot). For instance, a voter can cast a ballot with two ballot sheets, where one is valid and one is not (it is intentionally marked as invalid or violates some validation rules)

To validate a decoded ballot sheet, we consult the corresponding ballot specification. A decoded ballot sheet is discarded as invalid, if one of the following cases holds:

V1. The ballot sheet is explicitly marked as invalid.

V2. One or more *candidate rules* are violated, that is for some candidate the number of crosses for this candidate is bigger or smaller than required by the corresponding candidate specification; see CandidateSpec (Section **??**), fields `maxVotes` and `minVotes`.

V3. One or more *candidate list rules* are violated (see CandidateList (Section **??**), fields `maxVotesOnList`, `minVotesOnList`, `maxVotesForList`, `minVotesForList`), that is:

   (a) the total number of votes given to candidates on this list (not including the votes for the whole list) is not in the allowed range from `minVotesOnList` to `maxVotesOnList`,
   (b) the number of votes for the whole list (in the case of list voting) is not in the allowed range from `minVotesForList` to `maxVotesForList`.

V4. A *ballot sheet rule* is violated (see Core3Ballot (Section E.14), fields `minVotes` and `maxVotes`), that is the total number of votes (of all kinds) on the whole ballot sheet is not in the allowed range from `minVotes` to `maxVotes`.

Final counting. All the valid sheets, that is sheets from well-formed ballots which have not been not rejected according to the above rules, take part in the final counting. The procedure simply sums up all the votes for every candidate identifier (and/or candidate

list identifier, in the case of list voting) included in those sheets. Note that, in the ballot specification, candidate identifiers are mapped to human-readable descriptions, which allows one to produce a final result in a readable form.

B Additional Verification Tasks

B.1 Second Device Public Parameters

When the election system is deployed and bootstrapped, it creates the so called *second-device public parameters*, that is parameters of the voting process relevant for the ballot receipt signing/verification and ballot audit (cast-as-intende verification with a second device). These parameters include:

- the public election key (used to encrypt ballots),
- the receipt verification key (used by the main voting device and by the second device to check signatures on the ballot receipts issued by the voting system to the voter),
- the content of the ballots used in the election.

The e-voting system offers these public parameters, along with their fingerprint (the SHA-512 hash of the JSON representation of the second-device public parameters), for download via the Election Admin Panel. If a second device application is deployed (for the cast-as-intended ballot audit), such an application should be pre-configured with this fingerprint.

A tool for universal verifiability should offer the possibility of checking that a given file with the public parameters fingerprint is consistent with the content of the bulletin boards. This process is described below.

The format. A file downloaded via the EAP provides the aforementioned public parameters and the corresponding fingerprint in the following JSON format:

```
{
    "publicParametersJson": PUBLIC_PARAMETERS_JSON,
    "fingerprint": FINGERPRINT
}
```

where `PUBLIC_PARAMETERS_JSON` is a string with the JSON representation of the public parameters, as described below, and `FINGERPRINT` containins the SHA256 hash of the `PUBLIC_PARAMETERS_JSON` as a hexadecimal string.

The public parameters are of the form

```
{
    "publicKey": PUBLIC-ELECTION-KEY,
```

```
    "verificationKey": RECEIPT-VERIFICATION-KEY,
    "ballots": BALLOTS
}
```

where PUBLIC-ELECTION-KEY is the public election key (X.509 encoded and represented as a hexadecimal string),
RECEIPT-VERIFICATION-KEY is a (hexadecimal representation) of an RSA public key for checking signatures on ballot receipts, and BALLOTS is a list of ballots offered in the election.

For instance, the content of a downloaded file with public parameters can have the following format:


```
{
   "publicParametersJson":"{\"publicKey\":\"026ad8b0f428265c0e491856093a7b9b2ad9411da053bb844fe1f62b2af4a7cc93\"....
   "fingerprint":"979c947e10bf15d69d38595c4ae7f91b6214578264be099b3abbd5339117fedbd77c1596ec9d79769637d8701b528f....
}
```


Verification. The verification procedure should check that:

1. The fingerprint FINGERPRINT is in fact the SHA-256 hash of PUBLIC_PARAMETERS_JSON
2. The data in PUBLIC-ELECTION-KEY, RECEIPT-VERIFICATION-KEY, and BALLOTS is consistent with the content of the bulletin boards. That is, the verification procedure should make sure that:
   - The value PUBLIC-ELECTION-KEY is equal to the public key published on board keygen-electionKey-Polyas (Section D.6.2).
   - The value RECEIPT-VERIFICATION-KEY is equal to the verification key published on board BBox-ballotbox-key-CP (Section D.3.2).
   - The value BALLOTS is the list of ballots, as published on board registry (Section D.1.2), in the field `ballotStructures`.

B.2 Receipts

The voters have the option to save *ballot cast confirmations* (receipts), containing fingerprints of their encrypted ballots, signed by the election system. Such a receipt does not reveal the plaintext voter's choice (how the voter voted) and can therefore be safely handed out to third parties for audit.

A voter can download a ballot cast confirmation using the main voting application (right after the ballot cast step), as well as using the second device application, if the cast-as-intended mechanism with the second device is employed in the election.

The process can be organised in such a way that a voter can hand over their receipt to auditors (who carry out the universal verifiability procedure) in order to make sure that

their ballot is included in the final tally (that is in the verification package which is used to confirm correctness of the final tally). To support this process, a verification tool needs to be able to parse receipt files and make the required consistency checks, as detailed below.

Format of the receipt file. Receipts (ballot cast confirmations) are downloaded by the voters as simple PDF files containing a signed ballot fingerprint in the following format:

```
--BEGIN FINGERPRINT--
5a65b1192c8ddbc5a19ac6172c26b69561abdc5d5fdaffc8e47059980ab2cef1
--END FINGERPRINT--
--BEGIN SIGNATURE--
08ce02e615089992965080a2004e564908891ee4dafd0d2f7ef5e2590b9da988
0bcd7f621d84b01e75e88dfba6bcede7c479258b46287ffbcefae601ef8ea65d
3efac01842757921241bbb9e0a9e97e82aea67eee1857cc91edebf6736402c45
a284853c844aa4bd9f1404cb8a7faf9208016b1b6165a22bb36b3793f3467f72
3af83920d6383b37bcfb0bdf06916d459371dbd85322f1c41c50269d0c365440
c7f162f9cc3b507c35c92d4b22162a5683062f975fd6a65961aebc53af185af2
44cddce27236e6df513631258e116bb3a5947e1afe8354683a89a6354d8b4e15
1ca373fade2a86598ea11124af18b4485f9a215cc5053773b393e8b938429c59
--END SIGNATURE--
```

The first part contains the fingerprint of the encrypted ballot and the second one contains an RSA signature of this fingerprint, created by the election system. The verification application should extract this block from the PDF and apply the checks described below.

Note. The verification key for checking this signature is included in the second device public parameters, as defined above.

Verification. With a fingerprint and a signature extracted from the receipt file, the verification application should carry out the following steps:

1. Check that the included signature is a correct signature on the included fingerprint, using the *receipt verification key* published on board BBox-ballotbox-key-CP (Section D.3.2).

   If this check fails, report a warning (the provided file does not contain a correct signature of the election system). As, in this case, it may be impossible to resolve the issue (determine if the voter intentionally provided a fake receipt or if it was the voting application/second device application that generated invalid file), the election council may need to define a process to handle such cases.

2. If the above check has succeeded (the signature is correct), check that a ballot entry with the provided fingerprint is included in the ballot-box (Section D.1.3). In order to compute a fingerprint of an entry of the ballot box (of type BallotEntry (Section E.5)), apply the SHA256 hash function to the following data (using the conventions of Section A.1.2 for transforming the input data into the byte array to be hashed):

- the public label as a byte array (using the UTF-8 encoding), prepended with its length (as a 4-byte integer),
- the public credentials as a byte array, prepended with its length,
- the voter ID as a byte array (using the UTF-8 encoding) prepended with its length,
- the number of elements in the list of ciphertexts `ballot.encryptedChoice.ciphertexts` (as 4-byte integer) (see also Ballot (Section E.4)),
- the x and then the y components of each element of this list, represented as byte arrays prepended with their lengths,
- the number of elements in the list `ballot.proofOfKnowledgeOfEncryptionCoins`,
- the `c` and then the `f` component of each element of this list, represented as byte arrays prepended with their lengths,
- `proofOfKnowledgeOfPrivateCredential.c` and `proofOfKnowledgeOfPrivateCredential.f`, represented as byte arrays prepended with its length.

and take the hexadecimal representation of this hash.

For example, for the ballot entry


```
{
  "publicLabel" : "A",
  "voterID" : "03b36cfc60f1fa86a5826bb5377fe6ec123045df33652516101ae803cca57b4276",
  "ballot" : {
    "encryptedChoice" : {
      "ciphertexts" : [ {
        "x" : "0201df95626718539000d65a8049f2418328df95a61107357b96db2f5dc304b38b",
        "y" : "03692902cdfc6febebcd1d175859a6ea84018fe47c345f5e44fffddf97a492112f"
      } ]
    },
    "proofOfKnowledgeOfEncryptionCoins" : [ {
      "c" : "10188446347544979212343588959157565121623778797627805375625466440061560984940 2",
      "f" : "81174399198084050819276673106811016645306940417447240994496938329622216775741"
    } ],
    "proofOfKnowledgeOfPrivateCredential" : {
      "c" : "7444867864096567261070659423887179991345588536464902181904800065159483964442 6",
      "f" : "33276182696803028530168279419966710636866170477238993593368673329284148491287"
    }
  },
  "publicCredential" : "03b36cfc60f1fa86a5826bb5377fe6ec123045df33652516101ae803cca57b4276"
}
```


the digested bytes are


```
0000000141000000210 3b36cfc60f1fa86a5826bb5377fe6ec123045df336525
16101ae803cca57b427600000042303362333663666336306631666138366135
383236626235333737666536656331323330343564663333363532353136313 0
3161653830336363613537623432373600000001000000210201df9562671853
9000d65a8049f2418328df95a61107357b96db2f5dc304b38b00000021036929
02cdfc6febebcd1d175859a6ea84018fe47c345f5e44fffddf97a492112f0000
```

```
00010000002100e1409011d385f929f77566ed7bbf9e60677ed5946f7872931f
0ae564c4b1d63a0000002100b37714efd6c1c9cedaa0b3eb8c8d53c9aaec3e5e
1785073214b6623ae975183d0000002100a49875774197101742008 3c6c2ebd0
e6718c03d4e168dbd3f7859ed13894590a000000204991a6e74dc5ed7012c9aa
1a08d9e1b75a49e839abedff068b94419e5fe9f417
```

and the fingerprint is

```
534a6f16ae3f3e77e971d16bc4893c680824b5f16a882dfe299c629e33870dc3
```

C Format of the Verification Data

In this section we describe the data format of the data packet provided by the Election Provider to the Election Council at the end of the election. This data packet contains the content of all the bulletin boards used during the election process. This includes, in particular, the content of the ballot box, the registry board, the final (raw) result (decrypted ballots), and the intermediate data with zero-knowledge proofs.

The data packet is in the ZIP format. This file contains (among others), for each bulletin board with name *board-name*, a file named *board-name*`.json`. Such a file contains list of (JSON) records with the content explained below.

Note. Polyas routinely uses *secure bulletin boards* to store data (ballot box, the output of the intermediate steps, and the final result). Secure bulletin boards use digital signatures and hash chains in order to enhance integrity of the process (only new entries published by an authorised component are supposed to be appended to a bulletin board; no entries are supposed to be deleted or modified). Checking integrity of bulletin boards, by auditing the hash chains and digital signatures, is *not* subject of the verification task as described in this document (although the verification task can be extended in the future to include the verification of integrity of bulletin boards). Therefore, in the following description, we do not explain those elements of the data files which have to do with bulleting board integrity.

Each record in the file is for the form

```
{
    ...
    "c" : content
    ...
}
```

where the field labeled with `"c"` points to the content of the given record in JSON format (the remaining fields contain data related to bullleting board integrity and are, as mentioned, not discussed here). The type of the content depends on the bulletin board. For instance, for bulleting board *ballot-box*, content of each such a record is BallotEntry (Section **??**).

D Bulletin Boards

In this section we list all the bulletin boards used in the described instance of Polyas Core 3.0.

We distinguish *external* boards as a separate category, where such information is published as ballots (produced by voting clients) and registry record (uploaded by the Election Council). The content of the remaining boards is computed during the election process from the content of another boards.

A bulletin board can be blocked by so called *triggers*, which means that the system waits for some conditions before any data is published on the board. One type of triggers is related to election phase changes (some computations should only happen after some election phase is reached). Another type of a trigger is so-called guard trigger. In this case some computations wait till all the data from some selected boards is fully verified. For instance, decryption may have to be postponed until the complete mixing process has been verified.

Authorities

The operations are performed by the following *authorities*, where CP is the *election controller* owning the ballot box and other external boards:

- CP
- Polyas

Sub-components

The system consists of the following *sub-components* (logical gorups of boards):

- ballot
  Ballot pre-processing which flags and filters out incorrect ballots (for instance, ballots with invalid zero-knowledge proofs) and revoked ballots (if the election supports revocations) from the ballot box.
- BBox
  Boards belonging to the election controller (CP), related to the ballot box operations.
- decryption
  Ballot decryption
- final-guard
  Verification module (checking correctness of operations)
- keygen
  ElGamal key pair generation
- mixing
  Verifiable shuffle with encrypted ballots carried out by authorities Polyas
- pre-decryption-guard

Verification module (checking correctness of operations)

D.1 External boards

The input to the system is provides via the external boards listed below. This input is assumed to be provided by external control components. In particular, an appropriate component publishes ballots cast by voters on the board representing the ballot box.

Below, we list the external triggers and then provide details on the remaining external boards.

D.1.1 Triggers

- **init-trigger**
  Actors waiting for this trigger: keygen-secretKey-Polyas
- **registration-trigger**
- **tally**
  Actors waiting for this trigger: decryption-decrypt-Polyas
- **voting-trigger**

D.1.2 External board registry

Content: An entry of type `Registry<ECElement>`

Entry description: Registry entry containing, in particular, the ballot description and the list of registered voters

Example content


```
{
  "packetSize" : 400,
  "electionId" : "election-id",
  "desc" : "Test registry (1690801823430)",
  "ballotStructures" : [
    {
      "type" : "STANDARD_BALLOT",
      "id" : "0",
      "title" : {
        "default" : "ballot 0",
        "value" : { }
      },
      "lists" : [
        {
          "id" : "0-0",
          "title" : {
            "default" : "List 0",
```

```
          "value" : { }
        },
        "columnHeaders" : [
          {
            "default" : "List 0",
            "value" : { }
          }
        ],
        "candidates" : [
          {
            "id" : "0-0-0",
            "columns" : [ ],
            "maxVotes" : 1,
            "minVotes" : 0
          },
          {
            "id" : "0-0-1",
            "columns" : [ ],
            "maxVotes" : 1,
            "minVotes" : 0
          }
        ],
        "maxVotesOnList" : 999,
        "minVotesOnList" : 0,
        "maxVotesForList" : 999,
        "minVotesForList" : 0,
        "voteCandidateXorList" : false,
        "countCandidateVotesAsListVotes" : false
      }
    ],
    "showInvalidOption" : true,
    "showAbstainOption" : false,
    "maxVotes" : 999,
    "minVotes" : 0,
    "prohibitMoreVotes" : true,
    "prohibitLessVotes" : true,
    "calculateAvailableVotes" : false
  }
],
"voters" : [
  {
    "id" : "voter0",
    "publicLabel" : "0",
    "cred" : "0273a5e6f3549333e2c44856d786274d6037ae9911a7d72691a99265206bcaee52"
  },
  {
    "id" : "voter1",
    "publicLabel" : "0",
    "cred" : "02ed1a313178622e03dd1e339c568108a9be9059199dd3dca5870e10bd02392c55"
  },
  {
    "id" : "voter2",
    "publicLabel" : "0",
    "cred" : "03b36cfc60f1fa86a5826bb5377fe6ec123045df33652516101ae803cca57b4276"
  },
  {
    "id" : "voter3",
    "publicLabel" : "0",
    "cred" : "0266d3e8fb40919e8d426191ccac34439a0387cbc0199b3f09f6e1b4e8f93cc3ed"
  },
```

```
    {
      "id" : "voter4",
      "publicLabel" : "0",
      "cred" : "02ad64f815e369e50200fbf1573c8f0641b8406eeb7c2309f2ec5ce06c0351c6ac"
    },
    {
      "id" : "voter5",
      "publicLabel" : "0",
      "cred" : "028c9813a2a5bc33062005875bc12b410f9485f4224d7ca99d5d7f47b80b5dc713"
    },
    {
      "id" : "voter6",
      "publicLabel" : "0",
      "cred" : "029488096090843c9eb906d7676c26ee0f71a982fc194347fb35c37329667ac405"
    },
    {
      "id" : "voter7",
      "publicLabel" : "0",
      "cred" : "0301c99a59f0771d6551336e7b585d9394e687c2c9b454a5f018200799c691286f"
    }
  ],
  "revocationPolicy" : {
    "verificationKeys" : [ ],
    "threshold" : 0
  },
  "ballotSigningKey" : "30820122300d06092a864886f70d01010105000382010f003082010a028201010081e988ba4828a7aa200343d1a72cf4ba1be1f33e59672f178
}
```


D.1.3 External board ballot-box

Content: Sequence of entries of type `BallotPackage<ECElement>`

Entry description: Ballot box entry containing a list of individual (encrypted) ballots of type BallotEntry

Example content


```
- {
    "ballots" : [
      {
        "publicLabel" : "0",
        "voterID" : "voter0",
        "ballot" : {
          "encryptedChoice" : {
            "ciphertexts" : [
              {
                "x" : "022b5072bab556bf3c4bbe619009b7a551169725622d921924349d47faf38ded2d",
                "y" : "03dd18339d57519ea8c6e0919fdf2462acf5a2a662f06f9a15273cad77181d43a1"
              }
            ]
          },
          "proofOfKnowledgeOfEncryptionCoins" : [
            {
              "c" : "101966065510645828254625277125529590290617948118497246633917898776445258631298",
              "f" : "401040869257479144866599064907285332486313938249299535525362302140891419445 89"
            }
```

```
          ],
          "proofOfKnowledgeOfPrivateCredential" : {
            "c" : "6360383420628207896916647596310238174834836342805187865431612478160013798770 7",
            "f" : "669804011568425464520401716877902896127935282117861970389702608145595478998 82"
          }
        },
        "publicCredential" : "0273a5e6f3549333e2c44856d786274d6037ae9911a7d72691a99265206bcaee52"
      },
      {
        "publicLabel" : "0",
        "voterID" : "voter2",
        "ballot" : {
          "encryptedChoice" : {
            "ciphertexts" : [
              {
                "x" : "023a7693a62eb6c91904af43c6d56fcd0f37dc9b9aa3da4ebad6d61bf68c75685b",
                "y" : "03c0d87ebd4059ce17fe81aa594890ffe5298aa8aa9a892812696e2c0fecdb4afe"
              }
            ]
          },
          "proofOfKnowledgeOfEncryptionCoins" : [
            {
              "c" : "859552349052121209841548374615352285200593348916990078829967928580783591289 53",
              "f" : "108594512744852993959504323230746843866273612886032403335744352144996583730 419"
            }
          ],
          "proofOfKnowledgeOfPrivateCredential" : {
            "c" : "27707732107656503802413361147647612987275848460889726614659841603224508734213",
            "f" : "642845453330477178446681473805371846159363776916893620869010357647941531693 66"
          }
        },
        "publicCredential" : "03b36cfc60f1fa86a5826bb5377fe6ec123045df33652516101ae803cca57b4276"
      }
    ]
  }
- {
    "ballots" : [
      {
        "publicLabel" : "0",
        "voterID" : "voter4",
        "ballot" : {
          "encryptedChoice" : {
            "ciphertexts" : [
              {
                "x" : "029a5f954b72c4be04e3ec68cab72a09634642ccbd9b5d5288d561968317e6d446",
                "y" : "03d32ed47da8d443e7ce1d7e2dcd8d7d99b28211d2c80cf737396c1e916f51b195"
              }
            ]
          },
          "proofOfKnowledgeOfEncryptionCoins" : [
            {
              "c" : "372352091017620372820861742153182874103164509596172278500566984488950713517 87",
              "f" : "379854800751503965416190854321003058273400525323722501809599727126386177991 25"
            }
          ],
          "proofOfKnowledgeOfPrivateCredential" : {
            "c" : "998890785298845188054790755422889219229277237293492475305028318263363210727 6",
            "f" : "309819314810158956210060923414037481613957891493408745577420330639539753947 43"
          }
        },
```

```
      "publicCredential" : "02ad64f815e369e50200fbf1573c8f0641b8406eeb7c2309f2ec5ce06c0351c6ac"
    },
    {
      "publicLabel" : "0",
      "voterID" : "voter6",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "03b057df1bb0cf3197539344541a0441c1fa58e5b9e71bcb727197d7585a3b39ab",
              "y" : "02085d6bbccfd2e266de6ec94231ded63683f1f1108f31413827f0403e794c7e84"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "820063741852336409926538930057873362512814850892078108712766528906821666924 66",
            "f" : "37897034504007061903609375226663322453828078651604203325528054526301890391350"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "797535808517045198501387661567074550966266254053526727645260613507982151169 11",
          "f" : "837944373082104751390594286072691443072455882687317586603389249815888223777 24"
        }
      },
      "publicCredential" : "029488096090843c9eb906d7676c26ee0f71a982fc194347fb35c37329667ac405"
    }
  ]
}
```


D.1.4 External board revocation-authorisations

Content: Sequence of entries of type `RevocationTokenAuthorisation`

Entry description: An authorisation of a revocation token

Example content: (*no entries*)

D.1.5 External board revocations

Content: Sequence of entries of type `RevocationToken`

Entry description: A revocation token specifying voters whose ballots are to be revoked

Example content


```
- {
    "electionId" : "election-id",
    "voterIds" : [
      "voter1",
      "voter2"
    ]
  }
- {
```

```
  "electionId" : "election-id",
  "voterIds" : [
    "voter7"
  ]
}
```

D.2 Sub-component ballot

Ballot pre-processing which flags and filters out incorrect ballots (for instance, ballots with invalid zero-knowledge proofs) and revoked ballots (if the election supports revocations) from the ballot box.

D.2.1 Board ballot-flagged

Sub-component: ballot

Board name: ballot-flagged

Content: Entries of type `Packet<Annotated<BallotEntry<ECElement>>>`

Description: Ballots from the ballot box, along with an explicit annotation about their classification as correct, incorrect, or revoked. Published in packets.

Used by:

- ballot-filtered-out
- mixing-input-packets

Public input

- ballot-box — entries of type `BallotPackage<ECElement>`
- revocation-authorisations — entries of type `RevocationTokenAuthorisation`
- revocations — entries of type `RevocationToken`
- registry — an entry of type `Registry<ECElement>`
- keygen-electionKey-Polyas — an entry of type `PublicKeyWithZKP<ECElement>`

Example content

```
- {
    "values" : [
      {
        "ballot" : {
          "publicLabel" : "0",
          "voterID" : "voter0",
          "ballot" : {
            "encryptedChoice" : {
              "ciphertexts" : [
                {
                  "x" : "022b5072bab556bf3c4bbe619009b7a551169725622d921924349d47faf38ded2d",
                  "y" : "03dd18339d57519ea8c6e0919fdf2462acf5a2a662f06f9a15273cad77181d43a1"
```

```
          }
        ]
      },
      "proofOfKnowledgeOfEncryptionCoins" : [
        {
          "c" : "101966065510645828254625277125529590290617948118497246633917898776445258631298",
          "f" : "401040869257479144866599064907285332486313938249299535525362302140891419445 89"
        }
      ],
      "proofOfKnowledgeOfPrivateCredential" : {
        "c" : "6360383420628207896916647596310238174834836342805187865431612478160013798770 7",
        "f" : "669804011568425464520401716877902896127935282117861970389702608145595478998 82"
      }
    },
    "publicCredential" : "0273a5e6f3549333e2c44856d786274d6037ae9911a7d72691a99265206bcaee52"
  },
  "status" : "OK",
  "annotation" : ""
},
{
  "ballot" : {
    "publicLabel" : "0",
    "voterID" : "voter2",
    "ballot" : {
      "encryptedChoice" : {
        "ciphertexts" : [
          {
            "x" : "023a7693a62eb6c91904af43c6d56fcd0f37dc9b9aa3da4ebad6d61bf68c75685b",
            "y" : "03c0d87ebd4059ce17fe81aa594890ffe5298aa8aa9a892812696e2c0fecdb4afe"
          }
        ]
      },
      "proofOfKnowledgeOfEncryptionCoins" : [
        {
          "c" : "859552349052121209841548374615352285200593348916990078829967928580783591289 53",
          "f" : "108594512744852993959504323230746843866273612886032403335744352144996583730 419"
        }
      ],
      "proofOfKnowledgeOfPrivateCredential" : {
        "c" : "277077321076565038024133611476476129872758484608897266146598416032245087342 13",
        "f" : "642845453330477178446681473805371846159363776916893620869010357647941531693 66"
      }
    },
    "publicCredential" : "03b36cfc60f1fa86a5826bb5377fe6ec123045df33652516101ae803cca57b4276"
  },
  "status" : "REVOKED",
  "annotation" : "The voter voter2 has been revoked."
},
{
  "ballot" : {
    "publicLabel" : "0",
    "voterID" : "voter4",
    "ballot" : {
      "encryptedChoice" : {
        "ciphertexts" : [
          {
            "x" : "029a5f954b72c4be04e3ec68cab72a09634642ccbd9b5d5288d561968317e6d446",
            "y" : "03d32ed47da8d443e7ce1d7e2dcd8d7d99b28211d2c80cf737396c1e916f51b195"
          }
        ]
```

```
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "372352091017620372820861742153182874103164509596172278500566984488950713517 87",
            "f" : "37985480075150396541619085432100305827340052532372250180959972712638617799125"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "998890785298845188054790755422889219229277237293492475305028318263363210727 6",
          "f" : "309819314810158956210060923414037481613957891493408745577420330639539753947 43"
        }
      },
      "publicCredential" : "02ad64f815e369e50200fbf1573c8f0641b8406eeb7c2309f2ec5ce06c0351c6ac"
    },
    "status" : "OK",
    "annotation" : ""
  },
  {
    "ballot" : {
      "publicLabel" : "0",
      "voterID" : "voter6",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "03b057df1bb0cf3197539344541a0441c1fa58e5b9e71bcb727197d7585a3b39ab",
              "y" : "02085d6bbccfd2e266de6ec94231ded63683f1f1108f31413827f0403e794c7e84"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "8200637418523364099265389300578733625128148508920781087127665289068216669246 6",
            "f" : "37897034504007061903609375226663322453828078651604203325528054526301890391350"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "797535808517045198501387661567074550966266254053526727645260613507982151169 11",
          "f" : "837944373082104751390594286072691443072455882687317586603389249815888223777 24"
        }
      },
      "publicCredential" : "029488096090843c9eb906d7676c26ee0f71a982fc194347fb35c37329667ac405"
    },
    "status" : "OK",
    "annotation" : ""
  }
 ]
}
```


D.2.2 Board ballot-filtered-out

Sub-component: ballot

Board name: ballot-filtered-out

Content: Entries of type `String`

Description: All the ballots from the ballot box which have been filtered out (flagged as incorrect or revoked)

Used by: *none*

Public input

- ballot-flagged — entries of type `Packet<Annotated<BallotEntry<ECElement»>`

Example content

```
- "The voter voter2 has been revoked."
```

D.3 Sub-component BBox

Boards belonging to the election controller (CP), related to the ballot box operations.

D.3.1 Secret BBox-ballotbox-key-secret-CP

Computed by authority CP

Sub-component: BBox

Name: BBox-ballotbox-key-secret-CP

Content: An entry of type `SigningKeyPair`

Description: Secret (signing) RSA key used by the election system to sign ballot receipts

Used by:

- BBox-ballotbox-key-CP

D.3.2 Board BBox-ballotbox-key-CP

Computed by authority CP

Sub-component: BBox

Board name: BBox-ballotbox-key-CP

Content: An entry of type `VerificationKey`

Description: Public (verification) RSA key for checking signatures of the election system on the ballot receipts

Used by: *none*

Secret input

- BBox-ballotbox-key-secret-CP — an entry of type `SigningKeyPair`

Example content


```
"30820122300d06092a864886f70d01010105000382010f003082010a028201010081e988ba4828a7aa200343d1a72cf4ba1be1f33e59672f178db88dea49d1c7e154a60e
```


D.4 Sub-component decryption

Ballot decryption

D.4.1 Board decryption-decrypt-Polyas

Computed by authority Polyas

Sub-component: decryption

Board name: decryption-decrypt-Polyas

Content: Entries of type `MessageWithProofPacket<ECElement>`

Description: Result of the decryption of the input multi-ciphertexts (from the input ciphertext packets) along with zero-knowledge proofs of correct decryption

Generic description of the entry type: A packet of massages with a zero-knowledge proofs of correct decryption

Used by: *none*

Waits for (is blocked by)

- pre-decryption-guard-Polyas
- tally (a trigger)

Secret input

- keygen-secretKey-Polyas — an entry of type `BigInteger`

Public input

- mixing-mix-Polyas — entries of type `MixPacket<ECElement>`

Additional input for verification

- keygen-electionKey-Polyas — an entry of type `PublicKeyWithZKP<ECElement>`

Example content


```
- {
    "publicLabel" : "0",
    "messagesWithZKP" : [
      {
```

```
      "message" : "00000101",
      "proof" : [
        {
          "decryptionShare" : "0365e0f4a6459685e9d361e708cf4c38b35f6a06fb9aceb6c09d0961394a5999cc",
          "eqlogZKP" : {
            "c" : "513118543260835840464331952557689091467527135253772382902102831117044303310839",
            "f" : "80518035898711646999500546545873324892486561564661636242936907404456258667448"
          }
        }
      ]
    },
    {
      "message" : "00000101",
      "proof" : [
        {
          "decryptionShare" : "0268a41c9003bc7ed049c879c87650a5bb7fd1a139b06e714c151708ca60b3e263",
          "eqlogZKP" : {
            "c" : "545576674779067213912795806426865458025751470914280605511770976932361491911 28",
            "f" : "335811753727184121662610141754521400334644587486196707746331536882639227078 03"
          }
        }
      ]
    },
    {
      "message" : "00000101",
      "proof" : [
        {
          "decryptionShare" : "02ab453d431443982ee44fc82569db23d5a2d33ab76fd2e6304df5d9b571b14adc",
          "eqlogZKP" : {
            "c" : "124501611730011028658828422677914724767957949011443031098746095669152066123 04",
            "f" : "152573681185965019678834550018659025311670402158940367226611365192281296602 74"
          }
        }
      ]
    }
  ]
}
```


D.5 Sub-component final-guard

Verification module (checking correctness of operations)

D.5.1 Verification Component final-guard-Polyas

Computed by authority Polyas

Sub-component: final-guard

Board name: final-guard-Polyas

Blocked boards (components which wait for this verifier): *none*

Verified components:

- decryption-decrypt-Polyas

- tally

D.6 Sub-component keygen

ElGamal key pair generation

D.6.1 Secret keygen-secretKey-Polyas

Computed by authority Polyas

Sub-component: keygen

Name: keygen-secretKey-Polyas

Content: An entry of type `BigInteger`

Description: The secret (decryption) key corresponding to the desc key

Used by:

- keygen-electionKey-Polyas
- decryption-decrypt-Polyas

Waits for (is blocked by)

- init-trigger (a trigger)

D.6.2 Board keygen-electionKey-Polyas

Computed by authority Polyas

Sub-component: keygen

Board name: keygen-electionKey-Polyas

Content: An entry of type `PublicKeyWithZKP<ECElement>`

Description: The public desc key (used to encrypt ballots)

Used by:

- ballot-flagged
- mixing-mix-Polyas
- decryption-decrypt-Polyas

Secret input

- keygen-secretKey-Polyas — an entry of type `BigInteger`

Example content

```
{
  "publicKey" : "03f958e3d1832541f8bdd3f171128635a85899de0b2143da19575627b06fc7beb2",
  "zkp" : {
    "c" : "624225018664651038950561702739597795727896389302671797560681276884801390950 88",
    "f" : "106755395986867804727462515389388333944249790684922269117232904965084559531974"
  }
}
```


D.7 Sub-component mixing

Verifiable shuffle with encrypted ballots carried out by authorities Polyas

D.7.1 Board mixing-input-packets

Sub-component: mixing

Board name: mixing-input-packets

Content: Entries of type `MixPacket<ECElement>`

Description: Sequence of mix-packets containing ciphertexts to be shuffled, with the maximal packet size (that is the maximal number with ciphertexts in the packet) 400, obtained by grouping the ciphertexts (encrypted choices) from the input ballot entries. These mix-packets contain no zero-knowledge proofs.

Generic description of the entry type: Mix packet containing a list of ElGamal ciphertext plus, optionally, a zero-knowledge proof of correct shuffle

Used by:

- mixing-mix-Polyas

Public input

- ballot-flagged — entries of type `Packet<Annotated<BallotEntry<ECElement>>>`
- registry — an entry of type `Registry<ECElement>`

Example content


```
- {
    "publicLabel" : "0",
    "ciphertexts" : [
      {
        "ciphertexts" : [
          {
            "x" : "022b5072bab556bf3c4bbe619009b7a551169725622d921924349d47faf38ded2d",
            "y" : "03dd18339d57519ea8c6e0919fdf2462acf5a2a662f06f9a15273cad77181d43a1"
          }
        ]
      },
      {
```

```
      "ciphertexts" : [
        {
          "x" : "029a5f954b72c4be04e3ec68cab72a09634642ccbd9b5d5288d561968317e6d446",
          "y" : "03d32ed47da8d443e7ce1d7e2dcd8d7d99b28211d2c80cf737396c1e916f51b195"
        }
      ]
    },
    {
      "ciphertexts" : [
        {
          "x" : "03b057df1bb0cf3197539344541a0441c1fa58e5b9e71bcb727197d7585a3b39ab",
          "y" : "02085d6bbccfd2e266de6ec94231ded63683f1f1108f31413827f0403e794c7e84"
        }
      ]
    }
  ]
}
```


D.7.2 Board mixing-mix-Polyas

Computed by authority Polyas

Sub-component: mixing

Board name: mixing-mix-Polyas

Content: Entries of type `MixPacket<ECElement>`

Description: Sequence of mix packets obtained by shuffling each of the input mix packets and producing the appropriate zero-knowledge proof with correct shuffle.

Generic description of the entry type: Mix packet containing a list of ElGamal ciphertext plus, optionally, a zero-knowledge proof of correct shuffle

Used by:

- decryption-decrypt-Polyas

Public input

- mixing-input-packets — entries of type `MixPacket<ECElement>`
- keygen-electionKey-Polyas — an entry of type `PublicKeyWithZKP<ECElement>`

Example content


```
- {
    "publicLabel" : "0",
    "ciphertexts" : [
      {
        "ciphertexts" : [
          {
            "x" : "034851af152973ac3178f0854af8d6bc94c32885682d57bea1c06dcf276dba1822",
            "y" : "024dfa66c3f587b636fc1def0ae11d75761fccc4ac0583d3375724749215db9596"
          }
        ]
```

```
  },
  {
    "ciphertexts" : [
      {
        "x" : "037a272bd31992b7118aec92f8fdc41b1b4deebd974decfc6bb7420ee93f485dfc",
        "y" : "02ca6b0c6b5562fe31eb243f4e00cc62eb5bf4b3581d6d4f979b0ee79adb455266"
      }
    ]
  },
  {
    "ciphertexts" : [
      {
        "x" : "0310b9716fc868ca5d68245638b181f60bed917730ae1727aa3abaaf383fc8ba01",
        "y" : "0307acf3f4b6ceb73992542b1d57d95ec752340ed3977e45d5668837f277f5a900"
      }
    ]
  }
],
"proof" : {
  "t" : {
    "t1" : "03555f4c6b57ac23db08983230d4bc1c456dd3081ccace9f8fd6d7b52471c7b3ed",
    "t2" : "0355fe278c391e4a0248fc67232f02543864cf6a9115d1a966dd27edf4e7823735",
    "t3" : "037cafa78d65400023dc3568d61af04e688cc438e45dbc2f0a7aea61d47e3c199e",
    "t4y" : [
      "02461fc858fd2652db6044a34bd2f69a9243b8003d1b2e585b098c68cc7e47624d"
    ],
    "t4x" : [
      "02e259a37196cc95da858728009ed29356b25f984f6d7f5f99a4a16a8e885a0fe4"
    ],
    "tHat" : [
      "03d44e0989e39dd19c4bf0c6fe4d6be4ab135795558abd0e851c72084fcd0b37ae",
      "027a9c7d90c485695e1be0c03aa00f1380c5abceca87367ab1e6b41886016c11d2",
      "03b2368fe519d899b20e2f145ed921cbfe3d4e6a01f3ad6f819d938f59b8770b28"
    ]
  },
  "s" : {
    "s1" : "567839857072082765405559417360367095101203571578112598489026658907771921925 00",
    "s2" : "10185297645723447679282244982650889315189337819843933723200155188925230271 5727",
    "s3" : "537302215110751632484242076154558988533337865668818845929636942712095658567 76",
    "s4" : [
      "939229890213464864015176434235506340960556949519100969531253587583331596181 08"
    ],
    "sHat" : [
      "142847567249237253725472402692448065651137186827679545800140055302481253835 31",
      "856372647369083594204551463170188024508720795805911787028487448487064318131 60",
      "264969121003972672539076428284617991490477189508172900088187500229343161123 23"
    ],
    "sPrime" : [
      "685909082723252231923849525632719514710636850234214463036940403613900073731 15",
      "127177365507874973876897835094456744278045031649694953901116044978254580911 02",
      "351998966836456413446171971959292226902915550425167949349109265766072010859 61"
    ]
  },
  "c" : [
    "03472ef0cabe8562011189f22a83dbb67738a5dd8bba61f5ddc3d64b60913eddd5",
    "021eed06ebec2fa50ca8426bf76ccac249b4096415f92e6dbd50ef9e0abfbb1957",
    "031de596e5aaa79dbfcfbcbebd57d77239889d9a36a9ee5a0dada3a4d708196264"
  ],
  "cHat" : [
    "034c97cdf34adb191f97e688ebcfc87d31d15f440d0660b4bde98b4ac3ded9a26f",
```

```
      "02da00a4d60bc6e3afcce46a4d17455d01dfc4b1ae7ab31e6359a13702583db4d0",
      "0256e75edbabcf8afe7508df91c33137bfb3f99b76397d737e48a30c2e688b04fd"
    ]
  }
}
```


D.8 Sub-component pre-decryption-guard

Verification module (checking correctness of operations)

D.8.1 Verification Component pre-decryption-guard-Polyas

Computed by authority Polyas

Sub-component: pre-decryption-guard

Board name: pre-decryption-guard-Polyas

Blocked boards (components which wait for this verifier):

- decryption-decrypt-Polyas

Verified components:

- ballot-box
- ballot-filtered-out
- ballot-flagged
- init-trigger
- keygen-electionKey-Polyas
- mixing-input-packets
- mixing-mix-Polyas
- registry
- revocation-authorisations
- revocations

E Data Types

E.1 Annotated<B>

Type parameters: `B`

Description: A ballot with an additional annotation classifying the ballot as correct, incorrect, or filtered out (revoked)

Content:

- `annotation` : `String`
  (*mandatory*)
  If the ballot is not flagged as correct (that is if `status` is not `OK`), this field provides an additional information
- `ballot` : `B`
  (*mandatory*)
  Ballot entry (without annotations)
- `status` : `Status`
  (*mandatory*)
  Status annotation (correct, incorrect, or filtered out)

E.2 AutofillConfig

Content:

- `skipVoted` : `Boolean`
  (*mandatory*)
  If true autofill procedure will skip candidates that have already votes.
- `spec` : `AutofillSpec`
  (*mandatory*)
  Defines the autofill behavior. BALANCED means up to down in loops, TOPDOWN means left to right.

E.3 AutofillSpec

Enumeration type with values

- `BALANCED`
- `TOPDOWN`

E.4 Ballot<GroupElem>

Type parameters: `GroupElem`

Description: Encrypted ballot containing (encrypted) voter's choice and appropriate zero-knowledge proofs

Content:

- `encryptedChoice` : `MultiCiphertext<GroupElem>`
  (*mandatory*)
  Encrypted voter's choice (represented as a multi-ciphertext)

- `proofOfKnowledgeOfEncryptionCoins` : `List<DlogNIZKP.Proof>`
  (*mandatory*)
  Sequence of zero-knowledge proofs of knowledge of the random coins used to encrypt the voter's choice(one for each ciphertext in the multi-ciphertext `encryptedChoice`)
- `proofOfKnowledgeOfPrivateCredential` : `DlogNIZKP.Proof`
  (*mandatory*)
  Zero-knowledge proof of knowledge of the private credential

E.5 BallotEntry<GroupElement>

Type parameters: `GroupElement`

Description: Ballot entry containing: an encrypted ballot and voter's public credential, as the voter's unique identifier

Content:

- `ballot` : `Ballot<GroupElement>`
  (*mandatory*)
  Encrypted ballot
- `publicCredential` : `GroupElement`
  (*mandatory*)
  Voter's public credential
- `publicLabel` : `String`
  (*mandatory*)
  Public label of the voter; relevant only for elections where different voters vote for different ballots
- `voterID` : `String`
  (*mandatory*)
  The identifier of the voter

E.6 BallotPackage<GroupElement>

Type parameters: `GroupElement`

Description: Ballot box entry containing a list of individual (encrypted) ballots of type BallotEntry

Content:

- `ballots` : `List<BallotEntry<GroupElement»`
  (*mandatory*)

E.7 Block

Content:

- `data` : `Map<String, String>`
  (*mandatory*)
- `nodes` : `List<Node>`
  (*mandatory*)
- `type` : `String`
  (*mandatory*)
- `object` : `ObjectType`
  (*mandatory*)

E.8 CandidateList

Description: Specification of a candidate list (a list of candidates/voting options)

Content:

- `autofill` : `AutofillSpec`
  (*deprecated, optional*)
  If not null, the auto-fill behavior will be activated in the voter frontend application. This flag has no consequences on validation and counting.
- `autofillConfig` : `AutofillConfig`
  (*optional*)
  If present autofill is activate, with the provided configuration. This flag only has an effect on the behavior of the voter front-end, not on the validation and counting.
- `candidates` : `List<CandidateSpec>`
  (*mandatory*)
  List of candidate specifications
- `columnHeaders` : `List<I18n<String»`
  (*mandatory*)
  List of column headers
- `columnProperties` : `List<ColumnProperties>`
  (*optional*)
  Special properties for each column
- `contentAbove` : `Content<T>`
  (*optional*)
  Optional content to be displayed above the title
- `countCandidateVotesAsListVotes` : `Boolean`

(*deprecated, optional*)
If true, each vote for a candidate on this list is also counted as a vote for the list

- `derivedListVotes` : `DerivedListVotesSpec`
(*optional*)
Optional specification of derived list votes, where votes for candidates count as votes for the list
- `externalIdentification` : `String`
(*optional*)
Optional external identification of the list supplied by third party vendors like UniWahl4.
- `id` : `String`
(*mandatory*)
The identifier of the list, unique in the scope of the electionTemplate
- `maxVotesForList` : `Int`
(*mandatory*)
The maximal allowed number of votes for this list (crosses next to the list header). Non-zero value is used for elections with list voting.
- `maxVotesOnList` : `Int`
(*mandatory*)
The maximal allowed total number of votes (for regular candidates) on the list
- `maxVotesTotal` : `Int`
(*optional*)
The maximal allowed total number of votes (of both types: on and for the list)
- `minVotesForList` : `Int`
(*mandatory*)
The minimal required number of votes for this list (crosses next to the list header)
- `minVotesOnList` : `Int`
(*mandatory*)
The minimal required total number of votes (for regular candidates) on the list
- `minVotesTotal` : `Int`
(*optional*)
The minimal required total number of votes (of both types: on and for the list)
- `title` : `I18n<String>`
(*optional*)
The title (headline) of the candidate list (optional)
- `voteCandidateXorList` : `Boolean`
(*mandatory*)
If true, the voter is not allowed to place votes on the list and for the list at the same

time

E.9 CandidateSpec

Description: Specification of a candidate (voting option)

Content:

- `columns` : `List<Content<T»`
  (*mandatory*)
  The content of the consecutive columns describing the candidate; the number of columns should correspond to the number of the column headers in the enclosing `CandidateList`
- `externalIdentification` : `String`
  (*optional*)
  Optional external identification of the candidate supplied by third party vendors like UniWahl4.
- `id` : `String`
  (*mandatory*)
  The identifier of the candidate, unique in the scope of electionTemplate
- `maxVotes` : `Int`
  (*mandatory*)
  The maximal number of votes that can be given to the candidate
- `minVotes` : `Int`
  (*mandatory*)
  The minimal required number of votes that must be given to the candidate
- `writeInSize` : `Int`
  (*optional*)
  If non-null and non-zero, the field represents a write-in with the given size, where the content is used as a placeholder

E.10 Ciphertext<GroupElement>

Type parameters: `GroupElement`

Description: ElGamal ciphertext over the given set of group elements

Content:

- `x` : `GroupElement`
  (*mandatory*)
- `y` : `GroupElement`

(*mandatory*)

E.11 ColumnProperties

Description: Special properties for a column in a candidate list

Content:

- `hide` : `Boolean`
  (*mandatory*)
  Hide this column in the voter front-end

E.12 Content<T>

Description: Content with a value of some type `T` with internationalization.

Sealed class with the following subclasses:

- `Content.Text`
- `RichText`

E.13 Content.Text

Content:

- `value` : `I18n<String>`
  (*mandatory*)
  A value of type `T` with internationalization (possibly different values for different languages)
- `contentType` : `Type`
  (*mandatory*)

E.14 Core3Ballot

Description: Specification of a ballot content, validation and counting rules

Sealed class with the following subclasses:

- `Core3StandardBallot`

E.15 Core3StandardBallot

Description: Specification of a standard ballot content and ballot validation rules

Content:

- `calculateAvailableVotes` : `Boolean`
  (*mandatory*)
  If true, the voter front-end application displays the number of available votes. Does not affect validation and counting.
- `colorSchema` : `String`
  (*optional*)
  Optional color schema that the voter front-end application will use for this ballot. The color schema is the name of a CSS class that the voter front-end application knows.
- `contentAbove` : `Content`<T>
  (*optional*)
  Optional content to be displayed at the top of the ballot
- `contentBelow` : `Content`<T>
  (*optional*)
  Optional content to be displayed at the bottom of the ballot
- `externalIdentification` : `String`
  (*optional*)
  Optional external identification of the ballot supplied by third party vendors like UniWahl4.
- `id` : `String`
  (*mandatory*)
  Identifier of the ballot, unique in the scope of the electionTemplate
- `lists` : `List`<`CandidateList`>
  (*mandatory*)
  Sequence of candidate lists
- `maxListsWithChoices` : `Int`
  (*optional*)
  The maximum number of lists where the voter can place his/her choices. If `null`, there are no restrictions.
- `maxVotes` : `Int`
  (*mandatory*)
  The maximal total number of votes (crosses) of all kinds a voter can select in the whole ballot; ballots with bigger number of crosses will be rejected as invalid
- `maxVotesForCandidates` : `Int`
  (*optional*)
  The maximal allowed number of votes for candidates (as opposed to votes for lists) in the whole ballot

- `maxVotesForLists` : `Int`
  (*optional*)
  The maximal allowed number of votes for lists in the whole ballot
- `minVotes` : `Int`
  (*mandatory*)
  The lower bound on the total number of votes a voter can select in the whole ballot; ballots with smaller number of crosses will be rejected as invalid
- `minVotesForCandidates` : `Int`
  (*optional*)
  The minimal required number of votes for candidates (as opposed to votes for lists) in the whole ballot
- `minVotesForLists` : `Int`
  (*optional*)
  The minimal required number of votes for lists in the whole ballot
- `prohibitLessVotes` : `Boolean`
  (*mandatory*)
  If true, the client can't cast a vote with less crosses than required (by minVotes, minVotesForLists, minVotesForCandidates)
- `prohibitMoreVotes` : `Boolean`
  (*mandatory*)
  If true, the client can't cast a vote with more crosses than allowed (by maxVotes, maxVotesForLists, maxVotesForCandidates)
- `showAbstainOption` : `Boolean`
  (*mandatory*)
  If true, an abstention checkbox is shown
- `showInvalidOption` : `Boolean`
  (*mandatory*)
  If true, a checkbox is shown for explicitly marking the ballot as invalid
- `title` : `I18n<String>`
  (*mandatory*)
  Title of the ballot

E.16 DecryptionZKP.Proof<GroupElem>

Type parameters: `GroupElem`

Description: A non-interactive zero-knowledge proof of correct decryption

Content:

- `decryptionShare` : `GroupElem`
  (*mandatory*)
- `eqlogZKP` : `EqlogNIZKP.Proof`
  (*mandatory*)
  Non-interactive zero-knowledge proof of equality of discrete logarithms

E.17 DerivedListVotesSpec

Content:

- `variant` : `Variant`
  (*mandatory*)

E.18 DlogNIZKP.Proof

Description: Zero-knowledge proof of the knowledge of the descrete logarighm.

Content:

- `c` : `BigInteger`
  (*mandatory*)
- `f` : `BigInteger`
  (*mandatory*)

E.19 Document

Content:

- `data` : `Map<String, String>`
  (*mandatory*)
- `nodes` : `List<Node>`
  (*mandatory*)
- `object` : `ObjectType`
  (*mandatory*)

E.20 ECElement

Description: A point of the elliptic curve SecP256K1

E.21 EqlogNIZKP.Proof

Description: Non-interactive zero-knowledge proof of equality of discrete logarithms

Content:

- `c` : `BigInteger`
  (*mandatory*)
- `f` : `BigInteger`
  (*mandatory*)

E.22 I18n<T>

Type parameters: `T`

Description: A value of type `T` with internationalization (possibly different values for different languages)

Content:

- `default` : `T`
  (*mandatory*)
  The default value (used when there is no values specifed for the requested language)
- `value` : `Map<Language, T>`
  (*mandatory*)
  A mapping from languages to values of type `T`

E.23 Inline

Content:

- `data` : `Map<String, String>`
  (*mandatory*)
- `nodes` : `List<Node>`
  (*mandatory*)
- `type` : `String`
  (*mandatory*)
- `object` : `ObjectType`
  (*mandatory*)

E.24 Language

Description: Language code

Enumeration type with values

- `AR`
- `CZ`
- `DE`
- `DK`
- `EN`
- `ES`
- `FI`
- `FR`
- `HU`
- `IT`
- `NL`
- `NO`
- `PL`
- `RO`
- `ROU`
- `RU`
- `SE`
- `SK`
- `SVK`

E.25 Mark

Content:

- `data` : `Map<String, String>`
  (*mandatory*)
- `object` : `String`
  (*mandatory*)
- `type` : `String`
  (*mandatory*)

E.26 Message

Description: Sequence of bytes represented as a hexadecimal string

E.27 MessageWithProof<G>

Type parameters: G

Description: A massage along with a zero-knowledge proof of correct decryption

Content:

- auxData : Map<String, String>
  (*optional*)
  Data attached to the voter's ballot
- message : Message
  (*mandatory*)
  A decrypted message
- proof : List<DecryptionZKP.Proof<G»
  (*mandatory*)
  A zero-knowledge proof of correct decryption

E.28 MessageWithProofPacket<GroupElem>

Type parameters: GroupElem

Description: A packet of massages with a zero-knowledge proofs of correct decryption

Content:

- messagesWithZKP : List<MessageWithProof<GroupElem»
  (*mandatory*)
  List of decrypted messages with zero-knowledge proofs of correct decryption
- publicLabel : String
  (*mandatory*)

E.29 MixPacket<GroupElem>

Type parameters: GroupElem

Description: Mix packet containing a list of ElGamal ciphertext plus, optionally, a zero-knowledge proof of correct shuffle

Content:

- ciphertexts : List<MultiCiphertext<GroupElem»
  (*mandatory*)
  A list of ElGamal ciphertext
- proof : ShuffleZKProof<GroupElem>

(*optional*)
A zero-knowledge proof of correct shuffle (may be null)

- `publicLabel` : `String`
(*mandatory*)
The voter group

E.30 MultiCiphertext<GroupElement>

Type parameters: `GroupElement`

Description: List of ElGamal ciphertexts (over the given set of group elements); used to represent an encryption of a message which does not fit into one ciphertext

Content:

- `auxData` : `Map<String, String>`
(*optional*)
- `ciphertexts` : `List<Ciphertext<GroupElement»`
(*mandatory*)

E.31 Node

Sealed class with the following subclasses:

- `Block`
- `Inline`
- `Node.Text`

E.32 Node.Text

Content:

- `marks` : `Set<Mark>`
(*mandatory*)
- `text` : `String`
(*mandatory*)
- `object` : `ObjectType`
(*mandatory*)

E.33 ObjectType

Enumeration type with values

- `document`
- `block`
- `inline`
- `text`

E.34 Packet<A>

Type parameters: `A`

Content:

- `values` : `List<A>`
  (*mandatory*)

E.35 PublicKeyWithZKP<GroupElem>

Type parameters: `GroupElem`

Content:

- `publicKey` : `GroupElem`
  (*mandatory*)
  The public key
- zkp : `DlogNIZKP.Proof`
  (*mandatory*)
  A zero-knowledge proof of the knowledge of the private key corresponding to the public key (a proof of the knowledge of the discrete logarithm)

E.36 Registry<GroupElement>

Type parameters: `GroupElement`

Description: Registry entry containing, in particular, the ballot description and the list of registered voters

Content:

- `ballotSigningKey` : `String`
  (*mandatory*)
  Public key for ballot signing verification

- `ballotStructures` : `List<Core3Ballot>`
  (*mandatory*)
  Description of the ballots
- `desc` : `String`
  (*mandatory*)
  Description of the election
- `electionId` : `String`
  (*mandatory*)
  Unique identifier of the election
- `packetSize` : `Int`
  (*mandatory*)
  The maximal size of the mix packet
- `revocationPolicy` : `RevocationPolicy`
  (*optional*)
  Policy for ballot revocation. If null, the election does not support ballot revocations
- `voters` : `List<Voter<GroupElement»`
  (*mandatory*)
  List of voters with voter information

E.37 RevocationPolicy

Content:

- `threshold` : `Int`
  (*mandatory*)
- `verificationKeys` : `List<String>`
  (*mandatory*)

E.38 RevocationToken

Description: A revocation token specifying voters whose ballots are to be revoked

Content:

- `electionId` : `String`
  (*mandatory*)
  Election identifier (hash) specifying the election to which the revocation token applies
- `voterIds` : `List<String>`
  (*mandatory*)

A list of voters whose ballots are to be revoked

E.39 RevocationTokenAuthorisation

Description: An authorisation of a revocation token

Content:

- `publicKey` : `String`
  (*mandatory*)
  The public PGP key of the authority authorising the token, Base64-encoded
- `signature` : `String`
  (*mandatory*)
  The PGP signature of the authorising authority on the token, Base64-encoded
- `tokenFingerprint` : `String`
  (*mandatory*)
  The fingerprint of the authorised revocation token

E.40 RichText

Content:

- `value` : `I18n<Document>`
  (*mandatory*)
  A value of type `T` with internationalization (possibly different values for different languages)
- `contentType` : `Type`
  (*mandatory*)

E.41 ShuffleZKProof<GroupElement>

Type parameters: `GroupElement`

Description: Non-interactive zero-knowledge proof of correct shuffle (Wikstroem et al.)

Content:

- `c` : `List<GroupElement>`
  (*mandatory*)
- `cHat` : `List<GroupElement>`
  (*mandatory*)
- `s` : `ZKPs`

(*mandatory*)
The s-components of a zero-knowledge proof of correct shuffle

- `t` : `ZKPt<GroupElement>`
(*mandatory*)
The t-components of a zero-knowledge proof of correct shuffle

E.42 SigningKey

Content:

- `bytes` : `ByteArray`
(*mandatory*)

E.43 SigningKeyPair

Content:

- `signingKey` : `SigningKey`
(*mandatory*)
- `verificationKey` : `VerificationKey`
(*mandatory*)

E.44 Status

Enumeration type with values

- `OK`
- `INCORRECT`
- `REVOKED`

E.45 Type

Enumeration type with values

- `TEXT`
- `RICH_TEXT`

E.46 Variant

Enumeration type with values

- EACH_VOTE_COUNTS
- AT_MOST_ONE

E.47 VerificationKey

E.48 Voter<GroupElement>

Type parameters: GroupElement

Description: Information about an eligible voter

Content:

- aux : Map<String, String>
  (*optional*)
  The additional data attached to the voter
- cred : GroupElement
  (*mandatory*)
  The public credential of the voter
- id : String
  (*mandatory*)
  Voter's identifier
- publicLabel : String
  (*mandatory*)
  The public label (":" separated list of ballot sheet identifiers) assigned to the voter

E.49 ZKPs

Description: The s-components of a zero-knowledge proof of correct shuffle

Content:

- s1 : BigInteger
  (*mandatory*)
- s2 : BigInteger
  (*mandatory*)
- s3 : BigInteger
  (*mandatory*)
- s4 : List<BigInteger>
  (*mandatory*)
- sHat : List<BigInteger>

(*mandatory*)

- `sPrime` : `List<BigInteger>`
(*mandatory*)

E.50 ZKPt<GroupElement>

Type parameters: `GroupElement`

Description: The t-components of a zero-knowledge proof of correct shuffle

Content:

- `t1` : `GroupElement`
(*mandatory*)
- `t2` : `GroupElement`
(*mandatory*)
- `t3` : `GroupElement`
(*mandatory*)
- `t4x` : `List<GroupElement>`
(*mandatory*)
- `t4y` : `List<GroupElement>`
(*mandatory*)
- `tHat` : `List<GroupElement>`
(*mandatory*)

References

[1] PKCS #5: Password-Based Cryptography SpecificationVersion 2.1. ISSN: 2070-1721, https://tools.ietf.org/html/rfc8018.

[2] Standards for Efficient Cryptography 2 (SEC 2), 2010. Available at http://www.secg.org/sec2-v2.pdf.

[3] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, 2012.

[4] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International*

Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings, volume 740, pages 89–105. Springer, 1992.

[5] Rolf Haenni, Philipp Locher, Reto E. Koenig, and Eric Dubuis. Pseudo-code algorithms for verifiable re-encryption mix-nets. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2017.

[6] Thomas Haines. A Description and Proof of a Generalised and Optimised Variant of Wikström's Mixnet. Technical Report arXiv:1901.08371, arXiv, 2009. Available at https://arxiv.org/abs/1901.08371.

[7] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[8] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[9] Björn Terelius and Douglas Wikström. Proofs of Restricted Shuffles. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa*, volume 6055 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2010.

[10] Douglas Wikström. A commitment-consistent proof of a shuffle. *IACR Cryptology ePrint Archive*, 2011:168, 2011.

[11] Douglas Wikström. User Manual for the Verificatum Mix-Net Version 1.4.0. Verificatum AB, Stockholm, Sweden, 2013.