Polyas 3.0 E-Voting System
Variant for the GI 2019 Election

Version 0.9

February 28, 2019

Tomasz Truderung

Polyas GmbH

Contents

List of Algorithms

1 Introduction

This document describes a variant of *Polyas 3.0 E-Voting System* intended to be used in the GI elections starting from 2019.

We describe the structure of the protocol and the intended security properties. We also describe the tasks assigned to the Election Council (GI) 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, however, require participation of additional independent parties (division of roles); the role of such independent parties could be carried out by, for instance, designated entities within the GI. Participation of such independent parties involves some additional organisational effort. Moreover, to fully utilise the potential of the used cryptographic verifiability mechanisms, one would optimally use verification tools implemented by independent entities, which, again, requires some effort and time. This is why we established the following roadmap to enroll different aspects of security features in the coming years.

In the first step, for the **GI 2019 election**, we will provide a configuration of our e-voting system with the following features (we provide more details of this in Section 2):

- *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 tool is planned to be implemented by one or more independent parties; Polyas will provide a detailed documentation for such a verification tool by the end of February 2019.
- Voters' credentials (passwords) will be generated by the GI (a designated entity within the GI). The passwords will be then distributed to the voters using a secure printing facility. This mechanism provides a measure against so called *ballot stuffing* (unauthorized injecting of additional ballots to the ballot box).
  Polyas will provide a tool for credential generation, including the source code. Such a tool can be also independently implemented (by GI or a third party), based on the provided documentation.

In next steps we plan to implement additional measures in order to further strengthen the security features:

- In the **GI 2020 election**, *individual verifiability* will be addressed, which means that the voter should be able to verify that the ballot cast on his/her behalf contains, as expected, his/her indented choice and that this choice is not later altered or removed from the ballot box.

	system instance		
security feature	2019	2020	later
universal verifiability for tallying	yes	yes	yes
protection against removing/modifying ballots before tallying	no	yes	yes
cast as intended	no	yes	yes
distribution of trust for ballot secrecy	no	no	yes*

Figure 1: Summary of the security features.

These additional mechanism will not change the mixing and tallying components nor the universal verification procedure, as specified above (beyond perhaps some only minor changes).
A challenge of this step is that individual verifiability typically requires voters to carry out some additional steps and checks (which, by their nature, cannot be delegated to third parties, such as auditors). This makes the voting process somehow more complex and may require proper communication and education. One needs, therefore, to properly balance the security and usability aspects of such individual verifiability techniques.
In Section 3, we describe a solution we offer for individual verifiability based on use of a second device.

- Finally, in the **GI 2021 election or later**, we want to provide *stronger ballot secrecy guarantees* (which will remove the need to trust Polyas in this respect). This will be achieved by distribution of the secret election key amongst some number of independent parties (threshold decryption techniques) and distributed mixing (carried out, again, by some number of independent parties).
Although it is well known how to approach this issue from the cryptographic point of view, there are still some practical challenges to be addressed. The primary one is the robustness of the whole process (in general, distributed computations carried out by distinct, independent parties are more prone to instabilities due to, for example potential problems with network connectivity or misconfiguration). Therefore, well designed fall-back procedures must be established in order to guarantee uninterrupted election process.
Another challenging aspect is that, again, to fully utilise the security benefits of such distributed trust, one would want to use independently implemented components, which requires additional, potentially significant effort.

The summary of the security features in the planned configurations of the Polyas e-voting system for the GI elections is given in Figure 1.

In what follows, we, first, focus on the variant of the system for the GI 2019 elections. Then, in Section 3 we provide some more details concerning the future version.

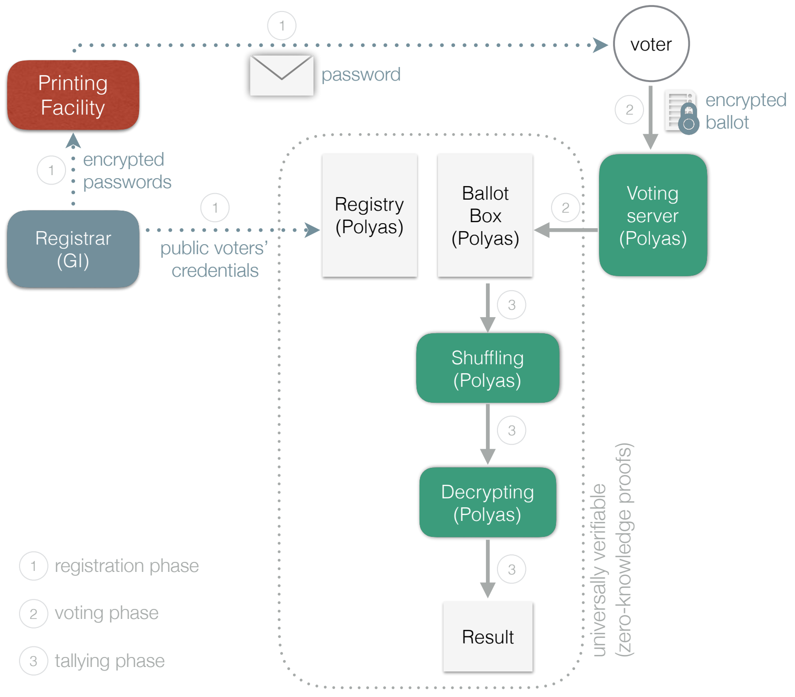

Figure 2: The structure of the system. Gray boxes represent bulletin boards. Ballot box is a bulletin board, where submitted ballots are published. Although not indicated on the picture, the intermediate results of shuffling (mixing) are also posted on bulletin boards and hence are subject to audit. In particular, verifiable shuffling and verifiable decryption produce zero-knowledge proofs of correctness of these operations.

2 Configuration for GI Election 2019

2.1 Overview of the Protocol

The protocol structure is depicted on Figure 2.

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 2), the Registrar (an entity designated by the GI) generates voters' passwords (private credentials) and the corresponding public credentials (private voters' credentials and the corresponding public credentials are is 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 transfered in an encrypted form to the secure printing facility which prints them out and distributes to the voters via 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 client code in the voter's browser), authenticates himself/herself to the voting platform, using his/her GI member number 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.

In the *tallying phase* (Step 3), the encrypted ballots collected in the ballot box 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

Polyas generates and maintain 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.

We note that the system is also designed to provide *participation privacy* with respect to the Election Council: under the assumption that the registration process has not been maliciously manipulated (which needs to be prevented by organisational measures), the system, and in particular the universal verifiability procedure, will not reveal to the Election Council which votes have cast their ballots (note that Polyas, in the current configuration, can learn identifier of voters who cast ballots).

2.3 Universal verifiability

As already mentioned, the complete tallying process (taking as its input the content of the ballot box and the content of the registration board) 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 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 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.

We will provide 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. Based on this specification, independent parties can then implement verification tools (which can be used by the Election Council). We will also provide 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 and distributed by an independent party called the *registrar* (a trusted entity designated by the Election Council): 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 printed and delivered by a secure printing facility.

During the voting phase, the voter's password is used, together with his/her GI member number, 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 hence 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).
- The printing facility does not know the association of the voters (to whom it sends passwords) with their GI member numbers. This provides some countermeasure against hypothetically dishonest printing facility trying to cast ballots on voter's behalf (the voting server must, of course, prevent brute-forcing of the member numbers).

Polyas will provide 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.

3 Configuration for GI Election 2020

As already explained, for the GI Election 2020, we plan to address the additional security guarantees related to *individual verifiability*. Individual verifiability is meant to make sure that (1) the encrypted ballot indeed contains the intended voter's choice and (2) the ballot has not been later altered nor removed from the ballot box. In this section, we sketch potential methods to extend the system with individual verifiability.

3.1 Receipts

To address Item (2), each voter, after having cast his/her ballot, will be issued a receipt (which can be printed or saved) which contains a signed hash of the encrypted ballot. This hash uniquely identifies the voter's encrypted ballot. Note that, because such a receipt only refers to an *encrypted* ballot, it is privacy preserving, i.e. the receipt does not reveal the voter's choice.

The Election Council may allow voters to check that their unaltered ballots (as uniquely identified by the receipt) are included, as expected, in the final content of the ballot box. This check is to be done after the election is over and is based on (a) the voter's receipt and (b) the data package provided by Polyas to the Election Council which contains the final content of the ballot box. The advantage of carrying out this check at this moment is that it explicitly makes sure that the ballot is included in the actual final content of the ballot box, the same content as the one the final election result is verified against.

It is up to the Election Council how this process is organized. The Election Council may consider allowing every voter to do this check (for instance, by submitting her receipt to a designated party) or allowing it only in case of justified doubts. It may be also useful to consider the idea of checking receipts of *designated auditors*, selected from the set of voters (a plus of such a solution is that the auditors could be trained in the details of the verification process).

3.2 Cast-as-intended

To address Item (1), we offer a solution based on the use of a second device (such as a mobile phone). The main device (the one used to cast the ballot), after the ballot has been cast, produces an encrypted proof of the plaintext content of the ballot. This encrypted proof is presented to the second device as a QR code. The second device (after re-authentication of the voter) decrypts and checks the proof. It then displays the voter's choice in a readable way, allowing the voter to make sure that it is, as expected, the choice that has been selected by the voter. Finally, the second device application allows the voter to save/print the voter's receipt, the same one as issued by the first device. By this we make sure that the ballot audited by the second device is the one the voter gets receipt for.

This audit protocol (see Figure 3) is organized in such a way that the QR code does not reveal the voter's choice (and therefore cannot be used as a receipt to sell the vote), unless the voter reveals his/her voting credentials (which would allow the vote buyer/coercer to cast a ballot on the voter's behalf in the first place).

The Election Council may consider the following options related the ballot audit application (run on the second device):

- a web application hosted by Polyas,
- a web application provided (with the source code) by Polyas, but hosted by the Election Council (such an application can be audited by an expert),
- a web application hosted by the Election Council and implemented by an independent entity (inside or outside the Election Council organisation),
- a native iOS/Android application implemented by an independent party.

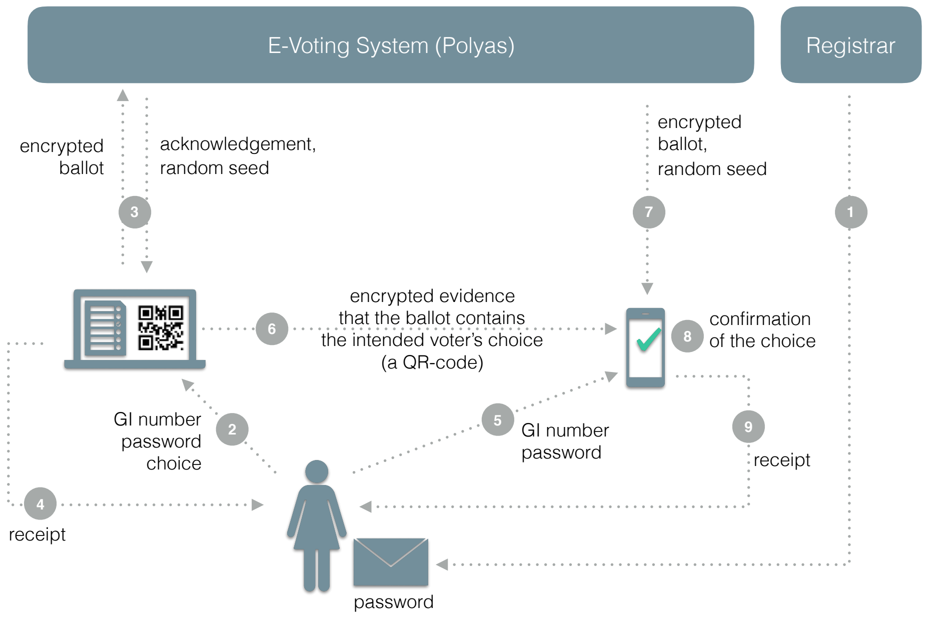
E-Voting System (Polyas)
Registrar
encrypted ballot
acknowledgement, random seed
3
encrypted ballot, random seed
7
1
6
encrypted evidence that the ballot contains the intended voter's choice (a QR-code)
8
confirmation of the choice
GI number password choice
2
5
GI number password
9
receipt
4
receipt
password

Figure 3: Protocol for the ballot audit with a second device. The voting application, after the ballot has been cast, displays a QR code containing the random encryption coin. This random coin is encrypted to prevent ballot buying (given in cleartext, it would reveal the voter choice). In order to audit the ballot, the voter uses her second device. The audit can be done only after successful re-authentication on the second device. This is because the key used to encrypt the random coin is derived from a random seed generated on the server side (the device has to be correctly logged in to obtain this seed, see Step 7), and from the secret voter's credential. After decrypting the random coin, the second device displays the plain text choice to the voter who can confirm that it is correct.

4 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 4. 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, while encrypted ballots, published on the ballot-box board, are created on the client side during the voting casting process and published

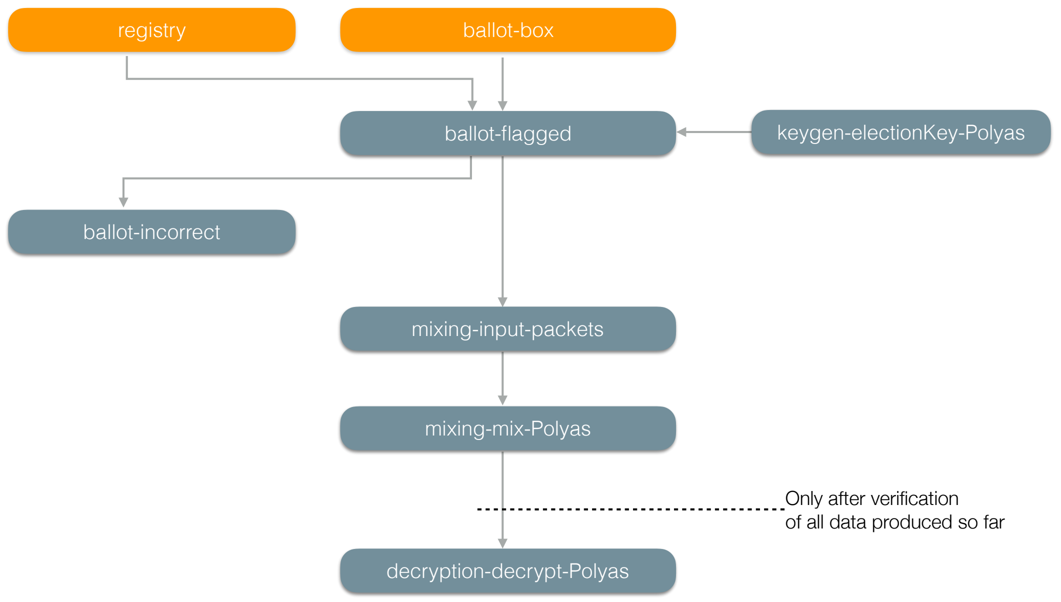

Figure 4: Bulletin boards of the system with indicated information flow between them. Orange boxes represent *external boards*; the content of all the remaining boards is computed (from the indicated input) by the Polyas Core 3.0 back-end.

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, possibly in a randomized way.

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 C.4.2)

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.

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 C.5.1) contains input mix packets, where all the correct ballots (that is ballots not flagged as incorrect) from the ballot-flagged (Section C.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 voter groups are used in a given election (that is if voters are assigned different groups, where voters from different groups fill up different ballots), then the above process is done independently for each voters' group (each packet contains encrypted ballots of voters from the same group, the order of ballots for each given group 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 C.5.1) are then cryptographically shuffled (packet-wise) and the shuffled (and re-encrypted) packets of ciphertexts are published on board mixing-mix-Polyas (Section C.5.2). 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 C.3.1).

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 ciphertext 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 C.3.1) 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 information. Appendix C contains technical details of all the involved bulletin boards. It includes also example data (taken from an actual, small system run). Appendix D lists and documents data types used in the system. Appendix B 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*.

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.

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 l above is 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 $B_1, \ldots, B_m$ and take the first l bytes of this sequence.

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).
```

Example: The first three numbers generated using Algorithm 2 for the initial seed "`xyz`" and $l = 520$ are

a_1 = `17325015042052204029009298204463087237056529450818255985939939131459420970011270206331380202180389681 0-9094917857329663184563374015879596834703721749398989648`

a_2 = `220740130366550343403153135551192297488969281760118350025926374262591406104614614292937677807282745046-19363005332069049797404744820588400037203799604910235 11`

a_3 = `188388958790351947735783851422395397995420134466568179836702319632872197572005215391358212215191378527-3222921786889836987731296728825119604809609410157987402`

a_4 = `142325984946721771118587479951560784284260278576787976662373628468020983270463839090041259719694875001-5976271793930713744890547611655064835165883323889981463`

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 981628745272234647160092861 52 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

  $$(\texttt{75788b8a22a04baad44c66ec80e86928597979bf1b287760ad4e3153293d613b},$$
  $$\texttt{664663757d16eff0b993ac12a1ba16ee4784ac08206b12be50f4d954d9d74c88})$$

  (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`)`.
- **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.

[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)`.

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.

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 $a_i$.
```

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

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

Algorithm 6: Decoding a message from a multi-plaintext

```
Input  : A positive number $q$ and a multi-plaintext $a_1, \dots, a_m$, 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 Truncate two left-most bytes and $k$ right-most bytes of $msg'$ to obtain $msg$.
4 return $msg$.
```

This multi-plaintext is mapped by Algorithm 6 back to *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` [1]. 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 [6]. 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.

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.
1 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.
2 for $w$ in $w_1, w_2, \ldots$ do
3     Define $x := w \mod p$.
4     Try to compute (using for instance the Tonelli–Shanks algorithm), the square
        root $\hat{y} = \sqrt{x^3 + ax + b} \mod p$.
5     if the square root $\hat{y}$ exists then
6         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).
7         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 $w$).
```

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. As already briefly explained (see Section 2.4), the general setting of this process is that 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.

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.

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

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,
- the public encryption key K of the election provider (Polyas).

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 the Base32 character set) containing, in the recommended variant, between 80 and 100 bits of entropy. 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:

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

Finally, the following values are computed:

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

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) 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; server will hash this derived password along with the user identifier 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 $address_i$, p_i (that is the voter's address and his/her password). This file will be encrypted using the public encryption key of the printing facility.
  The file is intended to be used in the following way. The printing facility will print the passwords and deliver them to the appropriate voters (with the given addresses).

- A file containing records of the form pk_i, $E_K(id_i,\ h_i)$, that is the voter's public credential plus, in encrypted form, the voter's identifier and his/her hashed password (recall that K denotes the public encryption key of the election provider). The records of this file will be sorted by the public keys (pk_i).
  The file is intended to be sent to the election provider and 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).

A.4 Key Generation

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 [7], where we use the strong Fiat-Shamir heuristic [2]. It is of the form $\pi = (c, f)$ and is created by drawing a random number a 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 C.4.2). Such a record is of the type PublicKey-WithZKP (Section D.18).*

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.

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

$pk =$ `03403091F3E81EE0E125FC33614DBA1ADBA569A3F7C05F9B36587054151508D490`

```
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.
```

The following proof (in JSON format)

```
{
  "c" : "62327941685486825449134997199289669684207465147565480140532634650865472277154",
  "f" : "51962687162358528709409258407636465178388247306669152965012697266119803118583"
}
```

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

A.5 Ballot Creation and Validation

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 by 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.

Note. *Ballots published on the ballot box are of type BallotEntry (Section D.3), which (in its subcomponent Ballot (Section D.2)) contains an encrypted ballot with the proof of knowledge of the encrypted coins and the proof of knowledge of the private credential. The ballots flagged during the ballot preprocessing, that is all the ballots enriched with the correctness information, are published on board ballot-flagged (Section C.2.1). As mentioned above, only the ballots which are marked as valid will be further tallied. Moreover, for each ballot marked as invalid, an entry is published on board ballot-incorrect (Section C.2.2) to explicitly protocol such an event.*

The verification task is supposed to check that this flagging process is done correctly (and report any incorrect entries). A ballot should be flagged as incorrect in two cases: (1) if there has been a ballot with the same voter identifier published before or (2) if the ballot is not well formed (does not have a valid zero-knowledge proof, as described below).

Note. *For the latter check, the public election key is used. This key is can be read from board keygen-electionKey-Polyas (Section C.4.2).*

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

- the plaintext voter's choice *msg*, given as a byte string (a sequence of bytes),
- the election public key $pk \in G$ (published on the board keygen-electionKey-Polyas, Section C.4.2),
- the public label l of the voter, given as string (recall that a public label is a public information assigned to the voter which determines which ballot a voter votes for; it is useful for elections where different voter groups are eligible to vote using different ballots),
- 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 [7]. 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).

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

Algorithm 9: Verification of a ballot entry

```
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 $e_i = (x_i, y_i) \in G^2$,
    • 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.
```

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":"55667612424127479016959768115758309554487545206887638059563287587298269617180",
     "f":"76750441957428754273366458063623429821774646529073833195390073367592941723801"},
    {"c":"87497191161142043606355252810633074518695312428888729285556397934908609418119",
     "f":"100800068275679310663391149138368437347267201966121012311937023355457387545611"}
  ],
  "proofOfKnowledgeOfPrivateCredential":{
    "c":"71296294066727390017142573272499110353651332475311228044572225570162122199458",
    "f":"93740793070444965834350731700811288291897877020039084234269069666765422852427"
  }
}
```

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. [8, 9] which is used, amongst others, in Verificatum [10]. 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 [4], 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 [5].

Note. *The packets to be shuffled are read from board mixing-input-packets (Section C.5.1). The shuffled packets are published on board mixing-mix-Polyas (Section C.5.2). 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 $h, h_1, \ldots, h_N \in G$.

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):

$$\begin{aligned} h &= \texttt{03F08FCB284F32B737E0529840334D481E055AD6AFA18AB91A3B02939EB19EB8DD} \\ h_1 &= \texttt{034A90A88BD2D3A92D7A29D19135F25536516D46FE4B8776C74B9E26D834FDA588} \\ h_2 &= \texttt{0365DB947FD33BE257599D9E0BD1513E6F7B3BBE6C9008382E22F4B527D3A39299} \\ h_3 &= \texttt{031E9073F6821FADE4307507F0D2756EFAAA4522FC15391390C943F4F4D9F32CE5} \end{aligned}$$

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 $\mathbf{a}_i$.

```
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 $\pi$ of the set $\{1, \dots, N\}$, randomness $\mathbf{r} \in \mathbb{Z}_q^N$
                       and randomness $R \in \mathbb{Z}_q^{N \times w}$, such that $\mathbf{c} = \mathit{Com}(M_\pi, \mathbf{r})$ 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

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

  $\mathcal{P}$ 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 $\mathcal{V}$:

      $\hat{c}_0 = h_1$,  $\hat{c}_i = h^{\hat{\mathbf{r}}_i} \hat{c}_{i-1}^{\mathbf{u}'_i}$  $(i \in \{1, \dots, N\})$    $t_1 = h^{\omega_1}$    $t_2 = h^{\omega_2}$    $t_3 = h^{\omega_3} \prod_{i=1}^N h_i^{\boldsymbol{\omega}'_i}$
      $\mathbf{t}_4 = \mathbf{ReEnc}_{g,pk}(\prod_{i=1}^N \mathbf{e}'^{\boldsymbol{\omega}'_i}_i, -\boldsymbol{\omega}_4)$    $\hat{\mathbf{t}}_i = h^{\hat{\boldsymbol{\omega}}_i} \hat{c}_{i-1}^{\boldsymbol{\omega}'_i}$  $(i \in \{1, \dots, N\})$

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

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

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

      $t_1 = (\prod_{i=1}^N \mathbf{c}_i / \prod_{i=1}^N h_i)^{-c} h^{s_1}$    $t_2 = (\hat{c}_N / h_1^{\prod_{i=1}^N \mathbf{u}_i})^{-c} h^{s_2}$    $t_3 = (\prod_{i=1}^N \mathbf{c}_i^{\mathbf{u}_i})^{-c} h^{s_3} \prod_{i=1}^N h_i^{\mathbf{s}'_i}$
      $\mathbf{t}_4 = \mathbf{ReEnc}_{g,pk}((\prod_{i=1}^N \mathbf{e}_i^{\mathbf{u}_i})^{-c} \prod_{i=1}^N \mathbf{e}'^{\mathbf{s}'_i}_i, -\mathbf{s}_4)$    $\hat{\mathbf{t}}_i = \hat{c}_i^{-c} h^{\hat{\mathbf{s}}_i} \hat{c}_{i-1}^{\mathbf{s}'_i}$
```

- 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).
- $\mathsf{Com}(M, \mathbf{r})$, for $M \in \mathbb{Z}_q^{N\times N}$ and $\mathbf{r} \in \mathbb{Z}_q^N$, is $(\mathbf{c}_1, \ldots, \mathbf{c}_n)$ 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 $\mathbf{ReEnc}_{g,pk}(\mathbf{e}_1, \mathbf{r}_1), \ldots, \mathbf{ReEnc}_{g,pk}(\mathbf{e}_w, \mathbf{r}_w)$.
- $\langle \mathbf{a}, \mathbf{b} \rangle$, 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 [5].

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.

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.

```
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 $\pi$ 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 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 $\mathbf{e}_1, \ldots, \mathbf{e}_N$ and ouput multi-cipherexts $\mathbf{e}'_1, \ldots, \mathbf{e}'_N$,
      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.
```

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 = `0300000000000000000000003B78CE563F89A0ED9414F5AA28AD0D96D6795F9C63`

The sequence $\mathbf{e}_1, \ldots, \mathbf{e}_N$ 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 $\mathbf{e}'_1, \ldots, \mathbf{e}'_N$ 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" : "69140316880887008429299433409534792696848419093216873680551974897144887159610",
    "s2" : "53131180115819471603649661885837848951867892970603491894778242682301254803439",
    "s3" : "66545885786812536207434432222640477995631070643429160379121299767828529923058",
    "s4" : [
      "49234149869250594959301514806927880761621594652382929864916536478567293576403",
      "44119365916638103229779040146256231483538164307118362944358852857821116505672",
      "86462011021256247510587356706438016595527168971353930254607186965833238087671"
    ],
    "sHat" : [
      "87579523046122018987817401009461204121160246014860600118901358882259284285635",
      "56314690457336236085857831679824894426738981904694725128459520767201510302300",
      "25753679853418879190896890932270665331640358030057655550457509094727662595142",
      "63915129513865911136597604340649908830039343581913211436624902429643358348225",
      "69854345023046456752350915483794001845993865628180922659193497065328728076315"
    ],
    "sPrime" : [
      "72366274572044528102669983191011032953991484588099121769246644882582908886523",
      "17352610575896663921431335332316518805152884681769238935143288047873148108374",
        "56595689515425780322167150059926269989757326187503888391850755958939583506650",
        "93208000741043741183848971908195921861581255731370504280744340908021922517310",
        "14357102550628836472011521008020057134171259373041892981299923206753076424152"
      ]
    }
  }
```

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

$$\begin{aligned}
u_1 &= 25423173261403838045780498659101929143374212024885961749677191123894345457203\\
u_2 &= 107163395173780552120959839682734915419294108435315434884329493449297702516989\\
u_3 &= 27527278399601879823598941265103560004203827776425685772980663219237052390097\\
u_4 &= 106382107453541024383519774022048119452768386444341851381719160092926187339847\\
u_5 &= 78025421855638301792439005550141533632218318123084187717794732643161239341502\\
c &= 14886957920142020425415970750713297044432709962075734803391029210025459699280
\end{aligned}$$

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 C.5.2). For each such packet a corresponding packet with decrypted messages is published on board decryption-decrypt-Polyas (Section C.3.1), 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 [3], where we use the strong Fiat-Shamir heuristic [2]. 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 C.3.1), where the decrypted binary messages along with zero-knowledge proofs of correct decryption are organized into records of type MessageWithProof (Section D.14) and then aggregated in packets of type MessageWith-ProofPacket (Section D.15) and published on board decryption-decrypt-Polyas (Section C.3.1).*

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":[
        {
            "decryptionShare" : "0220BF3495CDBDFCE2D10EB330AB72A56FC67B7D12BDB9270BB18D896089787FC6",
            "eqlogZKP" : {
              "c" : "86855984657246342025261681749033304437687691935040825490016247057942046977271",
              "f" : "97749087812374827457568318313550679277605194827170221699264620792657623830605"
            }
        }
    ]
 }
```

```
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 $B_i = \frac{x^{f_i}}{\alpha^{c_i}}$.
4     Compute $c'_i = \mathit{uniformHash}_q(g, x_i, pk, \alpha_i, A_i, B_i)$, where $\mathit{uniformHash}$ 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 $pk, \alpha_i$).
6 Compute, for each $i \in \{1, \ldots, n\}$, $d_i = \mathit{decode}(y_i/\alpha_i)$, where $\mathit{decode}$ 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 $\mathit{messageUpperBound}$, 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).
```

A.8 Final Ballot Tallying

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

Note. *Content of this section may be subject to some changes related to the way ballots are specified and encoded/decoded. In particular, if the election requires more complex ballots that can be expressed using the ballot specification described in this section, then the ballot specification will need to be extended in order to accommodate more complex ballots.*

Note. *Decrypted ballots in binary format (represented as hexadecimal strings are published on board decryption-decrypt-Polyas (Section C.3.1), where decrypted messages (along with additional proofs of correct decryption) are grouped in records of type MessageWithProofPacket (Section D.15). Every such a record includes a public label and a list of records of type MessageWithProof (Section D.14), where field 'message' contains the decrypted ballot. All these decrypted ballots constitute the input to the final counting.*

We describe now how decrypted ballots in the binary are decoded. Later we will describe how such decoded ballots are validated and used to count the final result.

The procedure for ballot decoding is presented in Algorithm 14. The result of this procedure determines how much votes a given encoded ballot assigns to different candidates, referred to by choice/candidate identifier. Additionally, as can be seen from the description of the algorithm, the encoded choices are organized in ballot sheets (voting lists) in a way corresponding to how ballots are presented to the voter during ballot casting.

Example. The decoding of the byte string `0x01070208020901` gives the following interpretation:

`0x01`	`0x07`	`0x02`	`0x08`	`0x02`	`0xff`	`0xff`	`0x09`	`0x01`	`0xff`	`0xff`
$m = 1$	$sid_1 = 7$ $f_1 = \text{false}$	$n_1 = 2$	$id_{1,1} = 8$	$v_{1,1} = 2$	—	—	$id_{1,2} = 9$	$v_{1,2} = 1$	—	—

The ballot contains one sheet ($m = 1$) which is not marked as invalid ($f_1 = \text{false}$) with sheet id 7 and with two entries ($n_1 = 2$): two votes for the candidate with identifier 8 and one choice for the candidate with identifier 9.

To further interpret decoded ballots, one needs to consult the content of the registry (Section C.1.1) board. This board contains, in particular, the specification of the ballot. Such a specification defines the structure and content of ballots, including candidate/choice lists and voting rules (for instance, how much selections a voter can make). We describe the content of the registry in more detail below.

Public labels and voting groups. To handle the case where different voters are eligible to vote for different set of ballots, we use the mechanism of *voter groups* and *public labels.*

```
Algorithm 14: Decoding of a decrypted ballot
  Input  : A binary content of the ballot (as a byte array)
  Output : A list of records of the form $\{sid_i, f_i, (v_{i,1}, id_{i,1}), \ldots, (v_{i,n_i}, id_{i,n_i})\}$
           where $sid_i$ is the identifier of a sheet, $f_i$ is a boolean flag indicating if the
           sheet has been explicitly marked as invalid, and $v_{i,j}$ is the number of
           votes given for the candidate with identifier $id_{i,j}$.
1 Read the first byte of the input to determine the number $m$ of sheets.
2 for $i = 1, \ldots, m$ do
3     Read next byte from the input; the first (left-most) bit $f_i$ of this byte
        determines if the $i$-sheet is explicitly marked as invalid; the remaining 7 bits
        encode the sheet identifier $sid_i$ of the $i$-th sheet;
4     Read next byte from the input to determine the number $n_i$ of cast choices on
        this sheet.
5     for $j = 1, \ldots, n_i$ do
6         Read next (unsigned) byte from the input to determine the identifier $id_{i,j}$
            of the choice/candidate.
7         Read next byte from the input to determine the number $v_{i,j}$ of
            crosses/votes for this choice/candidate; for compatibility reasons, we
            require that the first (left-most) bit of this byte is zero.
8         Read and ignore next two bytes from the input. These two bytes will be
            removed in the production version of the system.
```

Note: For elections without voter groups, the following explanation is not relevant; it is enough to note that in this case there is only one voter group and only one (corresponding) public label (all the ballots are labeled with the same public label).

In the general case, each voter can belong to some number of voting groups. The combination of the voting groups a given voter belongs to describes his/her voting rights (which ballots is he/she eligible to vote for). Such a combination is called a *public label* and is attached to the voters ballot (and then propagated throughout the tallying process). Note that ballots of voters with different public labels may have different lengths (because they encode choices for different ballot sheets).

As we will see, for each public label (combination of voters groups), the registry record contains an independent description of the ballot structure (ballot sheets with included voting options).

Registry. The record published on the registry board is of type Registry (Section D.19) and it contains, in its field `ballotStructures` a list of records of type BallotStructure (Section D.5) which describes ballots for a given *public label*. Each ballot specification (see BallotSpec (Section D.4)) contains, amongst others, list of sheets (see SheetSpec (Section D.20)) and each such a sheet contains, in particular, list of candidates/choices (see CandidateSpec (Section D.6). Decoded ballots refer to candidates/choices using identifiers defined here.

A record of type BallotStructure (Section D.5), as described above, with a given public label, specifies the set of sheets that a voter with this public label is eligible to vote for: this set contains all the sheets included in this record (that is all the sheets specified elements of type BallotSpec (Section D.4) listed in this record). Hence, the decoded ballot with a given public label, can contain sheets specified in such a record (with the same public label) and no other sheets.

See also an example content of the board registry (Section C.1.1).

Validation rules. Validation of ballots is done in two steps. First, malformed ballots are discarded. Second, sheets from well-formed ballots are validated against the election rules and invalid sheets are discarder. Consequently, only valid sheets from well-formed ballots contribute to the final result.

A precise definition of a malformed ballot is given below. Intuitively, a malformed ballot is not expected to be produced by the (honest) voter client. Therefore, presence of malformed ballot may suggest either a programming error, a non-standard behavior of a voter (who manipulates his/her browser), or a corrupt voters environment (browser problems, infected computer, etc.). On the contrary, invalid sheets can be produced by the voter client, if the voter makes some configuration of choices which is deemed invalid by the election rules.

A ballot is discarded as malformed if one of the following cases holds:

B1. The byte array (the ballot before decoding) is of wrong length (note that the expected length of unencrypted ballots for each public label is published on the registry board, see BallotStructure (Section D.5).

B2. It is impossible to parse it using Algorithm 14.

B3. The bytes not consumed by the decoding procedure (Algorithm 14) are not zero bytes (the expected way to encode ballots which do not require the full specified number of bytes is to pad them with zeros to the expected length).

B4. The ballot can be successfully parsed, but it contains wrong sheet or candidate identifiers, that is candidate/sheet identifiers not specified in the ballot specification, or candidate identifiers not belonging to the given sheet; see SheetSpec (Section D.20) and CandidateSpec (Section D.6).

B5. The ballot contains duplicated entries, that is more than one sheet with the same identifier or more than one candidate entry with the same identifier.

B6. One of the sheets is marked as explicitly invalid, although the corresponding sheet rules do not allow this; see SheetVotingRules (Section D.21).

If a ballot is well-formed, we validate independently all the sheets it contains. A sheet (from a well-formed ballot) is discarded as invalid, if one of the following cases holds:

S1. The sheet is explicitly invalid ($f_i = \mathsf{true}$ for some i).

S2. One or more candidate voting rules are violated, that is for some candidate the number of crosses for this candidate is bigger than specified in the corresponding candidate voting rules; see CandidateSpec (Section D.6) and CandidateVotingRules (Section D.7).

S3. One or more sheet voting rules are violated, that is, for some sheet on of the following conditions happens

   (a) the total number of selections made in this sheet is not in the allowed range as specified in the corresponding sheet voting rules, or
   (b) the total number of candidates/choices for which the voter gave at least one cross in not in the allowed range;

   See SheetSpec (Section D.20) and SheetVotingRules (Section D.21).

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 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 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 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*`.xml`. Such a file contains list of XML entries `<hcr>...</hcr>` 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.

In order to obtain the content of the board, we extract consecutive board entries from consecutive `<hcr>` elements (where 'hcr' stands for 'hash chain record'). Each such a `<hcr>` contains, in particular, an element `<c>...</c>` (where 'c' stands for 'content') which contains a JSON record of the form

```
{
    "paylod" : PAYLOAD_JSON
}
```

where `PAYLOAD_JSON` is a JSON representation of the content of the board entry. Note that this document specifies the expected type of entries for each bulletin board.

The above description applies to all the `<hcr>` entries except for the last one which contains a special, so-called, *seal* entry. This entry indicates that the secure bulletin board was sealed (closed) which means that from this point on no further entries could be added. The `<c>` element of this entry is of the form `<c>"sealedBoardName":"board-name"</c>`.

C 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 untill the complete mixing process has been verified.

Authorities

The operations are performed by the following *authorities*:

- 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 from the ballot box (for instance, ballots with invalid zero-knowledge proofs).
- decryption
  Ballot decryption
- keygen
  ElGamal key pair generation
- mixing
  Verifiable shuffle with encrypted ballots carried out by authorities Polyas.
- pre-decryption-guard
  Verification module (checking correctnes of operations)

C.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.

Triggers

- **init-trigger** — trigger
- **tally** — trigger

C.1.1 External board registry

Content: an entry of type `Registry<ECPoint>`

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

Example Content

```
- {
   "packetSize" : 400,
   "desc" : "Test desc 28.2.2019 11:03",
   "ballotStructures" : [
     {
       "publicLabel" : "[\"a\"]",
       "desc" : [
         {
           "id" : "8",
           "title" : {
             "value" : { },
             "default" : ""
           },
           "textAbove" : {
             "value" : { },
             "default" : ""
           },
           "textBelow" : {
             "value" : { },
             "default" : ""
           },
           "voterGroups" : [
             "a"
           ],
           "sheets" : [
             {
               "id" : "6",
               "title" : {
                 "value" : {
                   "DE" : "Candidates for the next Nobel Prize"
                 },
                 "default" : "Candidates for the next Nobel Prize"
               },
               "headers" : [
                 {
                   "value" : {
                     "DE" : "Name"
                   },
                   "default" : "Name"
                 },
                 {
                   "value" : {
                     "DE" : "Role in the company"
                   },
                   "default" : "Role in the company"
                 }
               ],
               "candidates" : [
                 {
                   "id" : "1",
                   "columns" : [
                     {
                       "value" : {
                         "DE" : "Jesica Bright"
                       },
                       "default" : "Jesica Bright"
                     },
                     {
                       "value" : {
                         "DE" : "President"
                       },
                       "default" : "President"
                     }
                   ],
                   "rules" : {
                     "maxVotes" : 2
                   }
                 },
                 {
                   "id" : "2",
                   "columns" : [
                     {
                       "value" : {
                         "DE" : "Amanda Foobar"
                       },
                       "default" : "Amanda Foobar"
                     },
                     {
                       "value" : {
                         "DE" : "Researcher"
                       },
                       "default" : "Researcher"
                     }
                   ],
                   "rules" : {
                     "maxVotes" : 2
                   }
                 },
                 {
                   "id" : "3",
                   "columns" : [
                     {
                       "value" : {
                         "DE" : "John Dow"
                       },
                       "default" : "John Dow"
                     },
                     {
```

```
                      "value" : {
                        "DE" : "Manager"
                      },
                      "default" : "Manager"
                    }
                  ],
                  "rules" : {
                    "maxVotes" : 2
                  }
                }
              ],
              "rules" : {
                "showInvalidOption" : true,
                "minVotes" : 0,
                "maxVotes" : 2,
                "minVotedCandidates" : 0,
                "maxVotedCandidates" : 3
              }
            }
          ]
        }
      ],
      "size" : 15
    }
  ],
  "voters" : [
    {
      "id" : "voter0",
      "publicLabel" : "[\"a\"]",
      "cred" : "0359DBA023D5C5CAE75C29B5DAF3BF593670E6D142A7CB8399070B4C59E718B57D"
    },
    {
      "id" : "voter1",
      "publicLabel" : "[\"a\"]",
      "cred" : "02A5150D7CB6381F53AE8D173E3B2FBD6F4B87EE44FA305C84A90254FFD223B779"
    },
    {
      "id" : "voter2",
      "publicLabel" : "[\"a\"]",
      "cred" : "02EFF7C8C71648848A9D4B6E860F7E1FAFCCB3CF852DA1BA370EBA6007D6D2DAD0"
    },
    {
      "id" : "voter3",
      "publicLabel" : "[\"a\"]",
      "cred" : "02B6AFF64EC9CC766E9D0E3611971815A91B1C7DC76A94BA26BF14669F89C7D929"
    }
  ]
}
```

C.1.2 External board ballot-box

Content: sequence of entries of type `BallotEntry<ECPoint>`

Entry description: Ballot entry containing: an encrypted ballot, voter's ID, and voter's public credential

Example Content

```
- {
   "publicLabel" : "[\"a\"]",
   "ballot" : {
     "encryptedChoice" : {
       "ciphertexts" : [
         {
           "x" : "0300DB20101F23F37F85E4551FE9A4938CA70E2F06BB9601498283F641A8B9A143",
           "y" : "0261BD9987CC92AD0D9D6F2583C32E7EEC13B8CC4C66B434467B745947CCC8804D"
         }
       ]
     },
     "proofOfKnowledgeOfEncryptionCoins" : [
       {
         "c" : "84581265262484548775678567277553504113415796762958443353234611261531837474723",
         "f" : "37356303802980064607963390289959128164087383697230454202859454476397977089124"
       }
     ],
     "proofOfKnowledgeOfPrivateCredential" : {
       "c" : "93297412636118950435855051236211451629458249832364498311943807943157667603876",
       "f" : "47037268692869402828496629464024017762736916498176880512997731184478357709909"
     }
   },
   "voterID" : "voter0",
   "publicCredential" : "0359DBA023D5C5CAE75C29B5DAF3BF593670E6D142A7CB8399070B4C59E718B57D"
 }
- {
   "publicLabel" : "[\"a\"]",
   "ballot" : {
     "encryptedChoice" : {
       "ciphertexts" : [
         {
           "x" : "024632A5A6E8BAD99BDEE3F7D46AE84DBDE408FD5550EB8752F5F8FDB5026B44E6",
           "y" : "03DA7B98F7BC57792E7421B794F0450D566891695433A7764C17541F20848CB8A1"
         }
       ]
     },
     "proofOfKnowledgeOfEncryptionCoins" : [
       {
         "c" : "40751354312229918239827079577218173526483788518859426450897468869092862837245",
         "f" : "34635584748949779630124118066253189137756243214502173996637525702393164699827"
       }
     ],
     "proofOfKnowledgeOfPrivateCredential" : {
       "c" : "67717431083811087554846421805804132020688642898541352192152016456779759094946",
       "f" : "48022766011222767024106565315792712675270540506223377149378508068716703668904"
     }
   },
   "voterID" : "voter1",
   "publicCredential" : "02A5150D7CB6381F53AE8D173E3B2FBD6F4B87EE44FA305C84A90254FFD223B779"
 }
- {
   "publicLabel" : "[\"a\"]",
   "ballot" : {
     "encryptedChoice" : {
       "ciphertexts" : [
         {
           "x" : "0336E6FB7D06EC82298C1C7720FEC4B30500A9612E0E22C43DAAF38859F007EFD9",
           "y" : "02D64477A8BFC4203BA91FB54C6EA53528085891A204C991FB1CF1175748912DC1"
         }
       ]
```

```
      },
      "proofOfKnowledgeOfEncryptionCoins" : [
        {
          "c" : "3526861848774947793432123734656657940576962336533139539647321914583227394650",
          "f" : "26281490724682671714770540235043260339201248944957966080119857505790393940595"
        }
      ],
      "proofOfKnowledgeOfPrivateCredential" : {
        "c" : "91878383302661426058529890113385204043816693939414565532565279718369963594739",
        "f" : "45650522851830309477610575917597117731218687204243941607066205393038992408478"
      }
    },
    "voterID" : "voter2",
    "publicCredential" : "02EFF7C8C71648848A9D4B6E860F7E1FAFCCB3CF852DA1BA370EBA6007D6D2DAD0"
  }
- {
    "publicLabel" : "[\"a\"]",
    "ballot" : {
      "encryptedChoice" : {
        "ciphertexts" : [
          {
            "x" : "02C98D6CA3A3FE7C5ABCF17751D0A1D57674A8EADAF5016561D5629CA9A76AB793",
            "y" : "03C547DF38357CE4F7397F1DD02E10372037D3DA2F167589ED84FC1ED915D68C58"
          }
        ]
      },
      "proofOfKnowledgeOfEncryptionCoins" : [
        {
          "c" : "77279196148654061260950090558645849342769182946616604538495775876633998756662",
          "f" : "39708634894719798369962226035071186923659153705875449187151034290251044553612"
        }
      ],
      "proofOfKnowledgeOfPrivateCredential" : {
        "c" : "22689504258852237895524364134760073145650527740824815856333273025963207567444",
        "f" : "15142425544111561768393897462706324721463979663175768049662854080732257039236"
      }
    },
    "voterID" : "voter3",
    "publicCredential" : "02B6AFF64EC9CC766E9D0E3611971815A91B1C7DC76A94BA26BF14669F89C7D929"
  }
```

C.2 Sub-component ballot

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

C.2.1 Board ballot-flagged

Sub-component: ballot

Board name: ballot-flagged

Content: entries of type `Annotated<BallotEntry>`

Description: Ballots (taken from the ballot box), where invalid ballots (for instance, ballots with invalid zero-knowledge proofs) are flagged as such

Generic description of the entry type: A ballot with additional correctness annotations

Used by:

- ballot-incorrect
- mixing-input-packets

Public input

- ballot-box — entries of type `BallotEntry<ECPoint>`
- registry — an entry of type `Registry<ECPoint>`
- keygen-electionKey-Polyas — an entry of type `PublicKeyWithZKP<ECPoint>`

Example output

```
- {
    "ballot" : {
      "publicLabel" : "[\"a\"]",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "0300DB20101F23F37F85E4551FE9A4938CA70E2F06BB9601498283F641A8B9A143",
              "y" : "0261BD9987CC92AD0D9D6F2583C32E7EEC13B8CC4C66B434467B745947CCC8804D"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "84581265262484548775678567277553504113415796762958443353234611261531837474723",
            "f" : "37356303802980064607963390289959128164087383697230454202859454476397977089124"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "93297412636118950435855051236211451629458249832364498311943807943157667603876",
          "f" : "47037268692869402828496629464024017762736916498176880512997731184478357709909"
        }
      },
      "voterID" : "voter0",
      "publicCredential" : "0359DBA023D5C5CAE75C29B5DAF3BF593670E6D142A7CB8399070B4C59E718B57D"
    },
    "ballotOk" : true,
    "annotation" : ""
  }
- {
    "ballot" : {
      "publicLabel" : "[\"a\"]",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "024632A5A6E8BAD99BDEE3F7D46AE84DBDE408FD5550EB8752F5F8FDB5026B44E6",
              "y" : "03DA7B98F7BC57792E7421B794F0450D566891695433A7764C17541F20848CB8A1"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "40751354312229918239827079577218173526483788518859426450897468869092862837245",
            "f" : "34635584748949779630124118066253189137756243214502173996637525702393164699827"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "67717431083811087554846421805804132020688642898541352192152016456779759094946",
          "f" : "48022766011222767024106565315792712675270540506223377149378508068716703668904"
        }
      },
      "voterID" : "voter1",
      "publicCredential" : "02A5150D7CB6381F53AE8D173E3B2FBD6F4B87EE44FA305C84A90254FFD223B779"
    },
    "ballotOk" : true,
    "annotation" : ""
  }
- {
    "ballot" : {
      "publicLabel" : "[\"a\"]",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "0336E6FB7D06EC82298C1C7720FEC4B30500A9612E0E22C43DAAF38859F007EFD9",
              "y" : "02D64477A8BFC4203BA91FB54C6EA53528085891A204C991FB1CF1175748912DC1"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
            "c" : "35268618487749477934321237346566579405769623365331395396473219145832273946 50",
            "f" : "26281490724682671714770540235043260339201248944957966080119857505790393940595"
          }
        ],
        "proofOfKnowledgeOfPrivateCredential" : {
          "c" : "91878383302661426058529890113385204043816693939414565532565279718369963594739",
          "f" : "45650522851830309477610575917597117731218687204243941607066205393038992408478"
        }
      },
      "voterID" : "voter2",
      "publicCredential" : "02EFF7C8C71648848A9D4B6E860F7E1FAFCCB3CF852DA1BA370EBA6007D6D2DAD0"
    },
    "ballotOk" : true,
    "annotation" : ""
  }
- {
    "ballot" : {
      "publicLabel" : "[\"a\"]",
      "ballot" : {
        "encryptedChoice" : {
          "ciphertexts" : [
            {
              "x" : "02C98D6CA3A3FE7C5ABCF17751D0A1D57674A8EADAF5016561D5629CA9A76AB793",
              "y" : "03C547DF38357CE4F7397F1DD02E10372037D3DA2F167589ED84FC1ED915D68C58"
            }
          ]
        },
        "proofOfKnowledgeOfEncryptionCoins" : [
          {
```

```
          "c" : "77279196148654061260950090558645849342769182946616604538495775876633998756662",
          "f" : "39708634894719798369962226035071186923659153705875449187151034290251044553612"
        }
      ],
      "proofOfKnowledgeOfPrivateCredential" : {
        "c" : "22689504258852237895524364134760073145650527740824815856333273025963207567444",
        "f" : "15142425544111561768393897462706324721463979663175768049662854080732257039236"
      }
    },
    "voterID" : "voter3",
    "publicCredential" : "02B6AFF64EC9CC766E9D0E3611971815A91B1C7DC76A94BA26BF14669F89C7D929"
  },
  "ballotOk" : true,
  "annotation" : ""
}
```

C.2.2 Board ballot-incorrect

Sub-component: ballot

Board name: ballot-incorrect

Content: entries of type `String`

Description: All the ballots from the ballot box which have been flagged as incorrect

Used by: *none*

Public input

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

Example output

C.3 Sub-component decryption

Ballot decryption

C.3.1 Board decryption-decrypt-Polyas

Computed by authority Polyas

Sub-component: decryption

Board name: decryption-decrypt-Polyas

Content: entries of type `MessageWithProofPacket<ECPoint>`

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

Secret input

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

Public input

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

Additional input for verification

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

Example output

```
- {
    "publicLabel" : "[\"a\"]",
    "messagesWithZKP" : [
      {
        "message" : "0106030101040102010401030104 01",
        "proof" : [
          {
            "decryptionShare" : "02ECD3543D9DED0DC1FC923F76449B7B3CC5379CCBB3782A7901EE643CB0707DA1",
            "eqlogZKP" : {
              "c" : "26628944004879402663420436016223643868236582177238198193072664881146579434427",
              "f" : "83892370686313503420574610116347253527581123837153369195624429866188194854619"
            }
          }
        ]
      },
      {
        "message" : "010602020104010301040100000000",
        "proof" : [
          {
            "decryptionShare" : "039F2B4B666BE6CB27611E78C7E4D2A1BAD4EB4E2A8ABBB62F499B4178B2205CD1",
            "eqlogZKP" : {
              "c" : "26078439359134908687177033215205165746474071067620243110997045774783223960495",
              "f" : "33644096566368268338459924715649346855863409846160246076998633732631440649034"
            }
          }
        ]
      },
      {
        "message" : "010601020204010000000000000000",
        "proof" : [
          {
            "decryptionShare" : "0278A96D4AC62DA462E152243D3381458A1B19295B4B4A6D1075035066E7B38143",
            "eqlogZKP" : {
              "c" : "91019719106061574768236943960320783697551057233599221523394204150433157433410",
            "f" : "16713684952933769802175362494721633701778922539611092499576418784876921463718"
          }
        }
      ]
    },
    {
      "message" : "018601010104010000000000000000",
      "proof" : [
        {
          "decryptionShare" : "0324A820BFE2808B1F20506E8FD37E88605ED198C8AC2DF15DACA81DE571E6D13C",
          "eqlogZKP" : {
            "c" : "89762155595477359023287228589170396058543477407460669394471221690031948499024",
            "f" : "61743059484767149624311946970958896307032558315567875311645600326472848383197"
          }
        }
      ]
    }
  ]
}
```

C.4 Sub-component keygen

ElGamal key pair generation

C.4.1 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

C.4.2 Board keygen-electionKey-Polyas

Computed by authority Polyas

Sub-component: keygen

Board name: keygen-electionKey-Polyas

Content: an entry of type `PublicKeyWithZKP<ECPoint>`

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

Generic description of the entry type: A public ElGamal key along with a zero-knowledge proof of the knowledge of the corresponding private key

Used by:

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

Secret input

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

Example output

```
{
  "publicKey" : "02D5E542831F8966A7B75CA8D19DF5C14BB83049508F56DCC669ACC421B887697D",
  "zkp" : {
    "c" : "95847909517928945282350875339622512722609734972191139559351906211981615173495",
    "f" : "108990507229961652523445352439575996837046102319681085106006878445195944003400"
  }
}
```

C.5 Sub-component mixing

Verifiable shuffle with encrypted ballots carried out by authorities Polyas.

C.5.1 Board mixing-input-packets

Sub-component: mixing

Board name: mixing-input-packets

Content: entries of type `MixPacket<ECPoint>`

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 conatining 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 `Annotated<BallotEntry>`

Example output

```
- {
    "publicLabel" : "[\"a\"]",
    "ciphertexts" : [
      {
        "ciphertexts" : [
          {
            "x" : "0300DB20101F23F37F85E4551FE9A4938CA70E2F06BB9601498283F641A8B9A143",
            "y" : "0261BD9987CC92AD0D9D6F2583C32E7EEC13B8CC4C66B434467B745947CCC8804D"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "024632A5A6E8BAD99BDEE3F7D46AE84DBDE408FD5550EB8752F5F8FDB5026B44E6",
            "y" : "03DA7B98F7BC57792E7421B794F0450D566891695433A7764C17541F20848CB8A1"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "0336E6FB7D06EC82298C1C7720FEC4B30500A9612E0E22C43DAAF38859F007EFD9",
            "y" : "02D64477A8BFC4203BA91FB54C6EA53528085891A204C991FB1CF1175748912DC1"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "02C98D6CA3A3FE7C5ABCF17751D0A1D57674A8EADAF5016561D5629CA9A76AB793",
            "y" : "03C547DF38357CE4F7397F1DD02E10372037D3DA2F167589ED84FC1ED915D68C58"
          }
        ]
      }
    ]
  }
```

C.5.2 Board mixing-mix-Polyas

Computed by authority Polyas

Sub-component: mixing

Board name: mixing-mix-Polyas

Content: entries of type `MixPacket<ECPoint>`

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

Example output

```
- {
    "publicLabel" : "[\"a\"]",
    "ciphertexts" : [
      {
        "ciphertexts" : [
          {
            "x" : "0306218C49F65C4C76E8DB7FA940ECD218F7ED8BBB1582D7385BC1E576E9B71C72",
            "y" : "02A7308707C5B563F6738FE13D9821A9C70AF17E3186C42650E45C683499BA11D3"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "02C93902FBDCF22D2D2320F1FD0EC66C483DD6B5804B2DC7FF0896E6F81AD291F5",
            "y" : "0285C08033DAE907E0AD29DBA2474E0972D77F016872BF5AACF7890BD8DC3E175D"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "02DD40A45326508CB0D21D5DE41204F0F9DAFF4963FDE9A54B6755449FA69261BF",
            "y" : "03B2862E84735DCA0DAA33B950C67B778242BF4029A22DF9A92C3809776C85A6A9"
          }
        ]
      },
      {
        "ciphertexts" : [
          {
            "x" : "0212EF2052C9996E22F9D65FF6025067D63593143AEDC2A46D55930422F9D337A4",
            "y" : "038C882724F98825CAA9332F1A7B451688D2A5E011B239FD4768BBF43AC60E1B1B"
          }
        ]
      }
    ],
    "proof" : {
      "c" : [
        "031E31AADAB06DEF865C8DB5D6DD86B4968B5766A20414A603B060BC01665A06F6",
        "0246A022C533F51F66B8FF9B6FC1DC6B5BF5A5139C4DDF0DFE9C5383851DDBB0DF",
        "0297464A378219095D7C4AADCB70257D35F9D0911F0AE1A8F9F80A817375AF077F",
        "034CA55F66B82A27D98927751575F100C75074793E3757EEE5E1DF2FFD46562EB3"
      ],
      "cHat" : [
        "02DB942F6924E1FF616DE2C0B0CF255EC6AFB34F34D7936DA901D81386A96E84D0",
        "025BF328D8A8CE4D58D18EC35BCB923E010602EF4B6E56CC43C7CA80BB30263E42",
        "02557AC6F1933732443F32C7CF768B8E2538ED896EE006EE069E884BBB3ECD29EF",
        "03149A75C04DF2F97327B5F74E30564178655A16C8F60499990DE14A42FE2BD3BA"
      ],
      "t" : {
        "t1" : "0323D14917B045445C75BA29295FDFEA7B7FDFABB4526B5A7C4D3CF282AB31BFF5",
        "t2" : "03E96A185E53366833742F48494DB0973D0A84D1396CB5E83B017F56E91156C44B",
        "t3" : "03C9F3C1779C42F54E38B218629C9FF6CC36F932A650A467AE0D7ED42BA3AB9D62",
        "t4y" : [
          "029FAFF4F3D5691F6474CC1754DB85604F47BF19D8269092625C7447AD86F5C213"
        ],
        "t4x" : [
          "0369FF41FEA2A1C0BA3E1A6B9B076E5EFAE7934B89E088D8971295F80EA9B77478"
        ],
        "tHat" : [
          "03C8D7E25814D7E7BCEC6014CFF6CD0620FF073EE937CE87ACD9600949D508A01E",
          "0362EC16096A3416FDCDE3988CF6A2A0C64676FE9240E8034C4BF591290AA49D37",
          "02BFB50F1596D1216C26731A058A67E3CACC72529D77850E56857EEAF4F1515307",
          "023690D24D2D9EA27EBCEC7D443FFBC0154F44FE5744AC114206D98D315B9BE852"
        ]
      },
      "s" : {
        "s1" : "62339249925413335843910369665385351287912771881145779764761099113517854878181",
        "s2" : "30748413606795003294601095688009183725593689922857016992538208873487596286105",
        "s3" : "52895336400272713042815587534721196224839660172023789650798965206629534070285",
        "s4" : [
          "81669810019647033439600933325255725724673886281581982127326796873899255443011"
        ],
        "sHat" : [
          "105199216580387642934601979357825746713762908166625120985084277639732080070884",
          "19472827403033190970215388240602885363895865356064557373086261682176183760869",
          "39779611068649402247719127828980685064028625898979849374950387512817295945813",
          "37043014292251452532495105813056340152193415700154909981077907409719944110532"
        ],
        "sPrime" : [
          "18920786016486922318323691866736240680043361781609938414411767294899264882095",
          "23781276893493794172345790227036492068299738978049583652599595207459495540650",
          "109704955859185845673021853054098118839877136827186072347651838633875396889629",
          "73389667923642793192811939863384601567181772819218378720830573570200686280807"
        ]
      }
    }
  }
```

C.6 Sub-component pre-decryption-guard

Verification module (checking correctnes of operations)

C.6.1 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-flagged
- ballot-incorrect
- init-trigger
- keygen-electionKey-Polyas
- mixing-input-packets
- mixing-mix-Polyas
- registry

D Data Types

D.1 Annotated`<B>`

Type parameters: `B`

Description: A ballot with additional correctness annotations

Content:

- `ballot` : `B`
  Ballot entry (without annotations)
- `ballotOk` : `Boolean`
  `true` if the ballot is correct, `false` otherwise
- `annotation` : `String`
  If the ballot is flagged as incorrect (that is if `ballotOk` is `false`), this field provides an additional explanation (why the ballot is incorrect)

D.2 Ballot`<GroupElem>`

Type parameters: `GroupElem`

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

Content:

- `encryptedChoice` : `MultiCiphertext<GroupElem>`
  Encrypted voter's choice (represented as a multi-ciphertext)
- `proofOfKnowledgeOfEncryptionCoins` : `List<DlogNIZKP.Proof>`

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`
  Zero-knowledge proof of knowledge of the private credential

D.3 BallotEntry<GroupElement>

Type parameters: `GroupElement`

Description: Ballot entry containing: an encrypted ballot, voter's ID, and voter's public credential

Content:

- `publicLabel` : `String`
- `publicCredential` : `GroupElement`
  Voter's public credential
- `ballot` : `Ballot<GroupElement>`
  Encrypted ballot
- `voterID` : `String`
  Identifier of the voter

D.4 BallotSpec

Description: Ballot configuration describing voter's choices and voting rules

Content:

- `id` : `String`
  Unique identifier of the ballot
- `sheets` : `List<SheetSpec>`
  List of ballot sheet specifications
- `textAbove` : `I18n`
  Text to be displayed above the ballot
- `textBelow` : `I18n`
  Text to be displayed below the ballot
- `title` : `I18n`
  The title of the ballot
- `voterGroups` : `List<String>`
  List of voter groups to which a voter must belong in order be eligible to vote for this ballot

D.5 BallotStructure

Description: Specification of ballots for a given public label (voters' groups)

Content:

- `publicLabel` : `String`
  The public label (corresponding to a set of voters' groups)
- `size` : `Int`
  Expected size of an unencrypted ballot in bytes
- `desc` : `List<BallotSpec>`
  List of specifications of ballots for which voters with the given public label (collection of voters groups) are eligible to voter for

D.6 CandidateSpec

Description: Specification of a single candidate/choice

Content:

- `columns` : `List<I18n>`
  Description of the candidate, given a list of elements to be displayed in consecutive columns
- `id` : `String`
  Unique identifier of a candidate/choice
- `rules` : `CandidateVotingRules`
  Rules for single candidate/choice

D.7 CandidateVotingRules

Description: Rules for single candidate/choice

Content:

- `maxVotes` : `Int`
  The maximal number of votes (crosses) the voter can give to this candidate

D.8 Ciphertext<GroupElement>

Type parameters: `GroupElement`

Description: ElGamal ciphertext over the given set of group elements

Content:

- `x` : `GroupElement`
- `y` : `GroupElement`

D.9 DecryptionZKP.Proof`<GroupElem>`

Type parameters: `GroupElem`

Content:

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

D.10 DlogNIZKP.Proof

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

Content:

- `c` : `BigInteger`
- `f` : `BigInteger`

D.11 EqlogNIZKP.Proof

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

Content:

- `c` : `BigInteger`
- `f` : `BigInteger`

D.12 I18n

Description: Text (message) with internalization (a mapping from language codes to messages)

Content:

- `default` : `String`
  Default message
- `value` : `Map<String, String>`
  Mapping from language codes to messages

D.13 Message

Description: Sequence of bytes represtened as a hexadecimal string

D.14 MessageWithProof`<G>`

Type parameters: `G`

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

Content:

- `message` : `Message`
  A decrypted message
- `proof` : `List<DecryptionZKP.Proof<G>>`
  A zero-knowledge proof of correct decryption

D.15 MessageWithProofPacket`<GroupElem>`

Type parameters: `GroupElem`

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

Content:

- `publicLabel` : `String`
- `messagesWithZKP` : `List<MessageWithProof<GroupElem>>`
  List of decrypted messages with zero-knowledge proofs of correct decryption

D.16 MixPacket`<GroupElem>`

Type parameters: `GroupElem`

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

Content:

- `proof` : `ShuffleZKProof<GroupElem>`
  A zero-knowledge proof of correct shuffle (may be null)
- `ciphertexts` : `List<MultiCiphertext<GroupElem>>`
  A list of ElGamal ciphertext
- `publicLabel` : `String`
  The voter group

D.17 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:

- `ciphertexts : List<Ciphertext<GroupElement>>`

D.18 PublicKeyWithZKP<GroupElem>

Type parameters: `GroupElem`

Description: A public ElGamal key along with a zero-knowledge proof of the knowledge of the corresponding private key

Content:

- `publicKey : GroupElem`
  The public key
- `zkp : DlogNIZKP.Proof`
  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)

D.19 Registry<GroupElement>

Type parameters: `GroupElement`

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

Content:

- `ballotStructures : List<BallotStructure>`
  Description of the ballots
- `packetSize : Int`
  The maximal size of the mix packet
- `voters : List<Voter<GroupElement>>`
  List of voters with voter information
- `desc : String`
  Description of the election

D.20 SheetSpec

Description: Specification of a single ballot sheet (list of candidate/choices)

Content:

- `candidates` : `List<CandidateSpec>`
  Description of choices (candidates)
- `headers` : `List<I18n>`
  Column headers
- `id` : `String`
  Unique identifier of a ballot sheet
- `rules` : `SheetVotingRules`
  Rules for a ballot sheet (list of candidates/choices) correctness
- `title` : `I18n`
  The title of the ballot sheet

D.21 SheetVotingRules

Description: Rules for a ballot sheet (list of candidates/choices) correctness

Content:

- `maxVotedCandidates` : `Int`
  The maximal number of candidates for which the voter gives at least one vote (cross)
- `maxVotes` : `Int`
  The maximal total number of selections (crosses) a voter can make on the sheet
- `minVotedCandidates` : `Int`
  The minimal number of candidates for which the voter gives at least one vote (cross)
- `minVotes` : `Int`
  The minimal required total number of selections (crosses) a voter has to make on the sheet
- `showInvalidOption` : `Boolean`
  A flag determining whether a voter can explicitly mark a ballot as invalid

D.22 ShuffleZKProof`<GroupElement>`

Type parameters: `GroupElement`

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

Content:

- `t` : `ZKPt<GroupElement>`
  The t-components of a zero-knowledge proof of correct shuffle
- `s` : `ZKPs`
  The s-components of a zero-knowledge proof of correct shuffle
- `c` : `List<GroupElement>`
- `cHat` : `List<GroupElement>`

D.23 Voter<GroupElement>

Type parameters: `GroupElement`

Description: Information about an eligible voter

Content:

- `publicLabel` : `String`
  The public label (voter's group) assigned to the voter
- `voterId` : `String`
  Voter's identifier
- `cred` : `GroupElement`
  The public credential of the voter

D.24 ZKPs

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

Content:

- `s1` : `BigInteger`
- `s2` : `BigInteger`
- `s3` : `BigInteger`
- `s4` : `List<BigInteger>`
- `sHat` : `List<BigInteger>`
- `sPrime` : `List<BigInteger>`

D.25 ZKPt<GroupElement>

Type parameters: `GroupElement`

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

Content:

- `t1` : `GroupElement`
- `t2` : `GroupElement`
- `t3` : `GroupElement`
- `t4x` : `List<GroupElement>`
- `t4y` : `List<GroupElement>`
- `tHat` : `List<GroupElement>`

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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`.

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

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

[8] 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.

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

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