Scytl

Innovating Democracy

Scytl sVote

Audit of the process with Control Components

Software version 2.1

Document version 3.1

Scytl - Secure Electronic Voting

STRICTLY CONFIDENTIAL

Table of contents

1 Introduction 9
1.1 Overview of the audit process 11
1.2 Organization 12
2 Data structures 13
2.1 SDM folder structure 13
2.1.1 The Authentication folder 16
2.1.2 The Extended Authentication 17
2.1.3 The Electoral Authority 17
2.1.4 The Voting Workflow 19
2.1.5 The Voter Material 19
2.1.6 The Vote Verification 20
2.1.7 The Election Information 24
2.2 Exported Ballot Box 29
2.3 Cleansed Ballot Box 35
2.4 Mixed and Decrypted Ballot Boxes in *CCM1*, *CCM2* and *CCM3* 35
2.5 Mixed and Decrypted Ballot Box in *CCM4* 40
3 File signature verification 43
3.1 Validate JSON files signature 43
3.1.1 Use a metadata JSON file with the same `.json` file name plus `.metadata` 43
3.1.2 Use a file with the same `.JSON` file name plus `.sign` 44
3.1.3 Store it in a field of the `.JSON` file 45
3.2 Validate CSV files signature 45
3.2.1 Use a metadata JSON file with the same `CSV` file name plus `.metadata` 45
3.2.2 Use a file with the same `CSV` file name plus `.sign` 45
3.2.3 Store it in the last line of the `CSV` file 45
4 Configuration validation 47
4.1 Certificates validation 53
4.2 Signatures validation 54
4.3 Control Components keys validation 55
4.3.1 Choice Return Codes encryption key pair 55
4.3.2 Mixing key pair 56
5 Vote decompression validation 57
6 Mixing and Decryption 58

6.1 Validation of the $CCM4$ output 58
6.2 Validation of the $CCM\mathbf{1}$, $CCM\mathbf{2}$ and $CCM\mathbf{3}$ outputs 59

7 Cleansing validation 60

8 Ballot Box Validation 62

8.1 Credential ID signing certificate validation 62
8.2 Signature validations 62
8.2.1 Vote Cast Return Code 63
8.2.2 Authentication token 63
8.2.3 Encrypted vote 64
8.2.4 Receipt 64
8.2.5 Verification Card Public Key 64
8.3 Proofs validations 65
8.3.1 Schnorr proof 65
8.3.2 Exponentiation proof validator 65
8.3.3 Plaintext equality proof validator 66
8.4 Vote validations 66
8.4.1 Vote hash validation 67
8.4.2 Vote format 67
8.4.3 Vote matches signing certificate 68
8.5 Codes Mapping Table validation 68
8.6 Consistent IDs validation 70
8.7 Voter information validation 71
8.8 Authentication token expiration time validation 71
8.9 Control Components Validation 71
8.10 Secure Logs validation 73

9 Authentication Validation 77

10 References 78

11 Appendix 79

11.1 Cryptographic primitives 79
11.1.1 Schnorr proof generator 79
11.1.2 Schnorr proof verifier 79
11.1.3 Exponentiation proof generator 80
11.1.4 Exponentiation proof verifier 81
11.1.5 Plaintext equality proof generator 82
11.1.6 Plaintext equality proof verifier 83

11.1.7 Decryption proof generator 84
11.1.8 Decryption Proof verifier 85
11.1.9 X.509 Certificate Validation 86
11.1.10 Mixing proof generator 88
11.1.11 Mixing proof verifier 103
11.1.12 Group Element generation 109
11.1.13 Commitment generation 109
11.1.14 ElGamal encryption 110
11.2 LDAP API 111
11.3 Coding and conversions 111
11.4 Data concatenation 112
11.5 Cryptographic algorithms 112
11.6 EV Solution Intellectual Property Rights Notice (the Notice) 112
11.6.1 Definitions 112
11.6.2 Copyright notice 113

List of figures

Figure 1 – SDM folder structure 13
Figure 2 - System keys folder 15
Figure 3 – Authentication folder 16
Figure 4 – ExtendedAuthentication folder 17
Figure 5 – ElectoralAuthorities folder 18
Figure 6 - public key JSON structure 18
Figure 7 – VotingWorkflow folder 19
Figure 8 – VoterMaterial folder 19
Figure 9 – VoteVerification folder 20
Figure 10 - Voter Choice Return Code/Vote Cast Return Code generation public key JSON 23
Figure 11 – ElectionInformation folder 25
Figure 12 – Ballot Box folder 27
Figure 13 – VoteSetID folder 29
Figure 14 - System certificate hierarchy 47
Figure 15 - Election Event certificate hierarchy 51
Figure 16 - Control Components Election Event certificate hierarchy 52

List of tables

Table 1 - Representation fields 31
Table 2 - System certificates details 50
Table 3 - Election Event certificates details 51
Table 4 - Control Components Election Event certificates details 52

List of files

File 1 - {adminBoardID}.pem 14
File 2 - platformRootCA.pem 14
File 3 - tenantCA.pem 14
File 4 - encryptionParameters.json 15
File 5 - authenticationContextData.json 16
File 6 - authenticationVoterData.json 17
File 7 - extendedAuthentication.csv 17
File 8 - electoralAuthority.json 18
File 9 - decryptionKey.json 19
File 10 - votingWorkflowContextData.json 19
File 11 - credentialData.csv 20
File 12 - voterInformation.csv 20
File 13 - codesMappingTablesContextData.csv 21
File 14 - verificationCardData.csv 21
File 15 - verificationCardSetData.json 21
File 16 - voteVerificationContextData.json 22
File 17 - derviedKeys.csv 22
File 18 - choiceCodeGenerationRequestPayload_{voteSetId}.json 23
File 19 - nodeContributions_{voteSetId}.json 24
File 20 - electionInformationContents.json 25
File 21 - ballot.json 26
File 22 - ballotBox.json 28
File 23 - ballotBoxContextData.json 28
File 24 - downloadedBallotBox.csv 29
File 25 - choiceReturnCodesComputationsJson 32
File 26 - ChoiceReturnCodesDecryptionJson 33
File 27 - voteCastCodeComputationsJson 34
File 28 - successfulVotes.csv 35
File 29 - failedVotes.csv 35
File 30 - CCM1, CCM2, CCM3 output 36
File 31 - voteEncryptionKey structure file 36

File 32 - voteSetId structure file 37
File 33 - previousVotes structure file 37
File 34 - shuffledVotes structure file 37
File 35 - votes structure file 37
File 36 - zkProof structure file 38
File 37 - Shuffle proof structure file 39
File 38 - commitmentParameters.json 40
File 39 - publicKey.json 41
File 40 - votes.csv 41
File 41 - votesWithProof.csv 42
File 42 - decompressedVotes.csv 42
File 43 - auditableVotes.csv 43
File 44 - metadata JSON file 43
File 45 - certificate in .PEM format 53

1 Introduction

This document provides a detailed description of how to verify that the election results accurately reflect the intention of legitimate voters. The audit of the system is possible due to the evidences (audit data) that the following components of the system produce during the whole election live cycle, starting at election configuration phase, continuing during the voting phase and finalizing at the end of the election counting phase (the details of the operations implemented during these phases are given in [1]):

- **Print Office:** Is the system component responsible for generating, printing and delivering the voting cards to the voters and for generating the election keys. Both the voting card generation and the election keys generation are done in physically isolated infrastructure.
- **Voting Server:** It authenticates the voter and receives, processes and stores in the Ballot Box the votes cast by them. From an architectural point of view, the voting server is implemented using different contexts. Each one of these contexts is in charge of executing a different part of the protocol**:**
    - **Voting Workflow Context**: Receives and manages client requests, contains the possible workflows per different Election Events and stores the status of the vote.
    - **Extended Authentication Context:** Participates in the first steps of the authentication process in case the system requires additional authentication values to start voting.
    - **Authentication Context:** Authenticates the voter in the system and performs authentication token validation.
    - **Election Information Context:** Stores the election information and the whole Ballot Box. performs vote and confirmation validations.
    - **Vote Verification Context:** Performs vote validations, stores the Choice Return Codes and the Vote Cast Return Code and retrieves them when requested.
    - **Voter Material Context:** Stores voter related materials.
- **Control Components:** There are two types of Control Components:
    - **Choice Return Codes Control Components** (CCR): They will implement the generation of the Choice Return Codes and the Vote Cast Return Code by using a distributed approach. These online components ($\mathrm{CCR}_1, \mathrm{CCR}_2, \mathrm{CCR}_3, \mathrm{CCR}_4$) will work in parallel and the results of their cryptographic operations will be combined to obtain the short Choice Return Codes and Vote Cast Return Code. These components will produce zero knowledge proofs as well as secure logs to give evidence that the encrypted votes stored in the Ballot Box have been validated and processed by them.
    - **Mixing Control Components** (CCM): These components implement the mixing and decryption of the votes during the counting process and will also be involved in the generation of the election key. By design, a Mixing can be implemented using several

Mix-nodes to shuffle and transform (re-encrypt) the votes in sequence. The approach will be based on implementing these mix-nodes using four Control Components, three of them online ($\mathrm{CCM}_1, \mathrm{CCM}_2, \mathrm{CCM}_3$) and one of them offline (CCM_4) executed in the Canton environment. With the aim of distributing the decryption process across them, each online mix-node performs a partial decryption after the mixing, using its own decryption key generated during the configuration. The last mix-node (CCM_4) decrypts the votes using an Electoral Authority key reconstructed in the Canton environment using a secret sharing scheme. These components will provide zero knowledge proofs that both the mixing and the decryption processes have been executed correctly, so the voters' privacy is protected.

- **Election Administrators:** They are responsible for generating the election configuration, verifying it, signing the results and publishing them. We distinguish between the **Administration Portal** that performs non-cryptographic operations (configure the ballot, define the Electoral Board members, etc.) and the **Administration Board** that assures the integrity and security of the voting process. This entity owns a key pair whose private key is shared among the Board members and is used to sign both the configuration, the outputs of the CCM_4 and the results. This key generation, key sharing and signature of the configuration are done in the Print Office environment. On the other hand, the Administration Board key reconstruction, the signature of the CCM_4 output and the signature of the results are done in the Canton environment.
- To implement some of the processes executed by the components mentioned above, a software component called **Secure Data Manager (SDM)** is used. The SDM is operated in the Print Office environment during the voting card generation and election key generation, and in the Canton Environment during the Administration Board and Electoral Board key reconstruction processes.

The Print Office needs to interact with the Choice Return Codes Control Components during the generation of the voting cards and with the online Mixing Control Components during the generation of the election keys. In addition, it also needs to interact with the Administration Portal to obtain the election configuration (e.g., candidate names). However, the Print Office is an environment designed to be implemented offline for enforcing its security (i.e., it is considered as a trusted component in the abstract security model). For this reason, another module is used as a bridge between the offline environments and the online ones. This bridge module also uses the SDM software component but does not perform any operation neither on the input nor on the output, and the integrity is preserved since the data is signed by the corresponding component. (e.g., Control Component).

During the counting phase, the reconstruction of the Electoral Board and the Administration Board key needed by the CCM_4 Control Component to decrypt the votes and signing the output information in the Canton environment, is done using the SDM software component.

As a result of the execution of the SDM in each environment and after the interaction with the online components using the bridge, a folder structure is created containing all the configuration, the Ballot Box and the output of the mixing and decryption processes.

For simplicity, from now on the bridge use of the SDM will be omitted and the explanation will refer directly to the interaction of the SDM module functions with the online components. In addition, it will be assumed that when there is a reference to SDM in the configuration phase, is the software component executed in the Print Office environment, and in the counting phase, is the software component executed in the Canton environment with the CCM_4.

The audit data produced by the system components mentioned above is located in the **Global Bulletin Board**, that is implemented as a distributed system, meaning that the information stored in it comes from different sources (local Bulletin Board) and repositories.

- The **Ballot Box** where the encrypted votes and their proofs are stored. Voting Server and Control Components are keeping a local Ballot Box of all the votes that are processed by the solution.
- The **Secure Logger** that registers all the actions that takes place in each entity by producing immutable logs that are protected by means of cryptographic mechanisms, ensuring that nobody can manipulate the entries stored in the log without being detected. The information stored in the log could be used to recognize any inconsistency in the votes cast and recorded in the Ballot Box. All the components of the solution have a Secure Logger of the transactions.
- The folder structure created to store the configuration and the output of the mixing and decryption processes.

1.1 Overview of the audit process

To ensure the integrity of the data processed through different voting system components, and that these processes are accurate and fair, the following auditing processes need to be performed:

- Ensure that the configuration generated in the SDM is the configuration used during the voting phase and that has not been altered after it has been generated. This can be easily verified since the Administration Board signs all the configuration generated in the SDM. Therefore, verifying the signature with the AB certificate, it can be ensured that this data has not been altered since its generation.
- Ensure that all the encrypted votes stored in the Ballot Box have been cast during the voting phase by voters that are in the electoral roll. This can be verified checking that all the votes stored in the Ballot Box correspond to valid authentication tokens generated during the authentication phase.

- Ensure that all the votes that are part of the tally correspond to votes that have been validated by the voters. This can be verified checking that the encrypted votes at the input of the counting process are only those that have been confirmed by the voters.
- Ensure the integrity of the Ballot Box, that is, no votes are deleted after they are cast, and no votes are added without being processed by the Control Components. This can be verified using the Secure Logs generated by the Control Components.
- Ensure that the output of each Choice Return Code Control Component has not been altered during transportation. Since the outputs are signed by the CCRs and stored in the Ballot Box, it is possible to validate that they have not been modified after their generation.
- Ensure that the output of each Mixing Control Component is the input of the following and that data have not been altered during transportation. Since each component digitally signs its output data, verifying the signature of a component input data with the previous component digital certificate, it can be ensured that this data has not been altered since its generation or processing by such previous component.
- Ensure that the behavior of each component is the expected. The ways in which we can verify the correct operation of a component may vary:
    - Cleansing component applies some public rules over the input votes. Therefore, they can be audited by applying such rules in the same input using independent software and checking that the output generated is the same.
    - Mixing Control Components use sensitive information, such as a private permutation of votes or private keys, to perform the mixing and decryption operations. This private information cannot be given to an audit application to reproduce the same output, since it could break the voters' privacy. For this reason, both the mixing and decryption processes generate mathematical proofs such that the correct behaviour of the component can be assured just validating the proofs and without using sensitive information.

1.2 Organization

This document is organized as follows:

- In section 2.1 we define which is the folder structure generated by the SDM and which is the information stored inside each folder.
- In section 3 we explain how to validate the signatures computed over the different types of files generated by the system.
- The following sections explain how to audit the configuration phase (section 4), the vote decompression operation (section 5), the mixing and decryption processes (section 6) and the cleansing (section 7).

- The validation of the Ballot Box is explained in section 8 and in section 9 how to validate that the votes in the Ballot Box have been cast by authenticated voters.
- Finally, the Appendix contains the description of several cryptographic primitives and some information common to all validations (e.g., how to concatenate data).

2 Data structures

In this section the audit data generated by the system components is presented, how it is stored and where it can be found. The generation of secure logs by the components has been omitted, since there will be a direct reference to them whenever a validation is performed.

The explanation covers first the folder structure generated by the Secure Data Manager, with a description of the files stored in each one of the folders and how the information is displayed inside.

Then, more details are provided on how the information is stored in the file that contains the Ballot Box and which is the output of the cleansing.

Finally, there is a differentiation between the output of the three online Mixing Control Components and the output of the offline one, what it contains and in which format.

2.1 SDM folder structure

Inside the `config` folder, the SDM generates as many folders as Election Events are currently running. The name of each of these folders is the corresponding `election_event_id`.

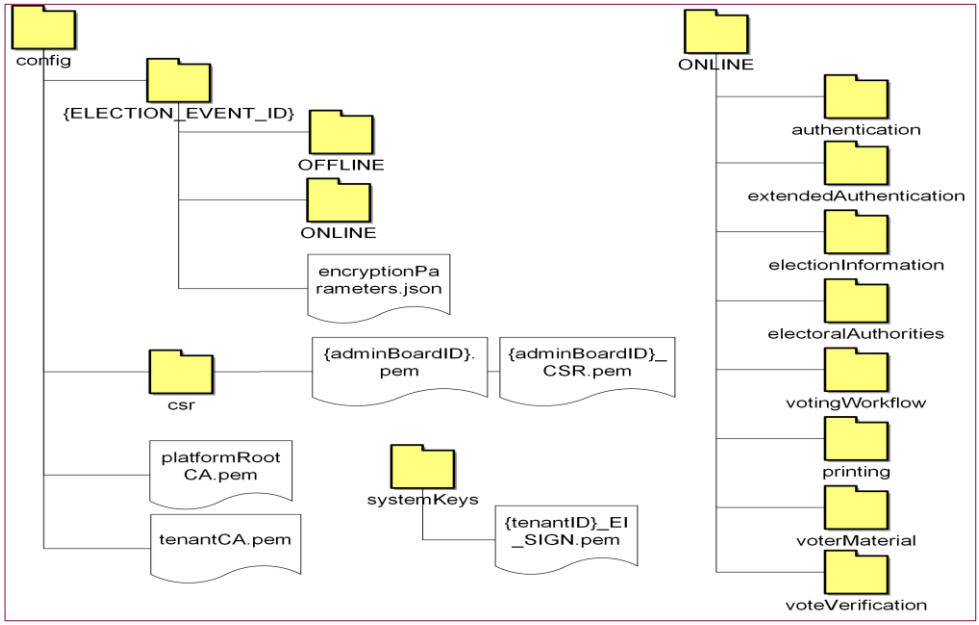


Figure 1 – SDM folder structure

Additionally, it generates the `csr` folder that contains the Administration Board digital Certificates in `PEM` format.

```
-----BEGIN CERTIFICATE-----
MIIDlDCCAnygAwIBAgIUDK2MyRFavMfrVbocJRewzVXOyr0wDQYJKoZIhvcNAQEL
BQAwXzEWMBQGA1UEAwwNVGVuYW50IDEwMCBDQTEWMBQGA1UECwwNT25saW5lIFZv
dGluZzEVMBMGA1UECgwMT3JnYW5pemF0aW9uMQkwBwYDVQQHDAAxCzAJBgNVBAYT
.......
-----END CERTIFICATE-----
```

File 1 - {adminBoardID}.pem

The `config` folder also contains the Platform Root CA and the Tenant CA:

```
-----BEGIN CERTIFICATE-----
MIIDlDCCAnygAwIBAgIUQDW64EpSXgpmKPFWUzDckHwm5OcwDQYJKoZIhvcNAQEL
BQAwXzEWMBQGA1UEAwwNVGVuYW50IDEwMCBDQTEWMBQGA1UECwwNT25saW5lIFZv
dGluZzEVMBMGA1UECgwMT3JnYW5pemF0aW9uMQkwBwYDVQQHDAAxCzAJBgNVBAYT
.......
-----END CERTIFICATE-----
```

File 2 - platformRootCA.pem

```
-----BEGIN CERTIFICATE-----
MIIDcDCCAligAwIBAgIVAI+oAu6TX79tz84yGef2FBh0xANeMA0GCSqGSIb3DQEB
CwUAMF8xFjAUBgNVBAMMDVNjeXRsIFJvb3QgQ0ExFjAUBgNVBAsMDU9ubGluZSBW
b3RpbmcxFTATBgNVBAoMDE9yZ2FuaXphdGlvbjEJMAcGA1UEBwwAMQswCQYDVQQG
.......
-----END CERTIFICATE-----
```

File 3 - tenantCA.pem

The `systemKeys` folder contains some of the keys generated during the system configuration process (we refer the reader to [1] for more details). These keys are those used to protect the integrity of the election keys.

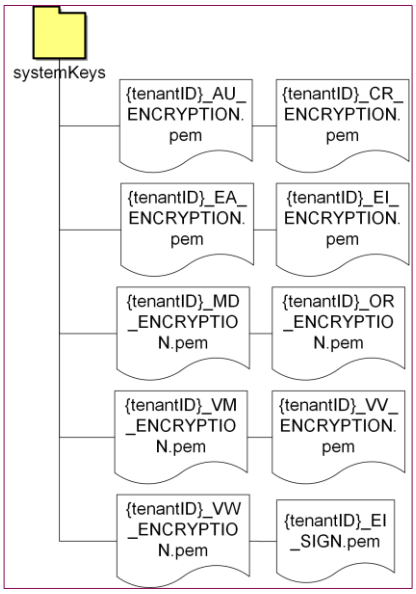


Figure 2 - System keys folder

Given one Election Event, two folders are generated.

- The **offline** folder contains information to be used during the configuration process of the election, but it will not be used outside the offline environment (Print Office).
- On the other hand, the **online** folder contains all the information that will be exported to be used during the election: voting cards, authentication information, election configuration, etc. and the information of the counting process. In fact, the online folder contains:
    - One folder for each of the contexts in the voter portal: `authentication`, `electionInformation`, `votingWorkflow`, `voterMaterial`, `voteVerification`, `extendedAuthentication`
    - One folder for the information that is going to be printed (the voting cards).
    - Another for the information regarding the electoral authorities.
    - Finally, the `encryptionParameters` file contains the values that define the mathematical group for the ElGamal encryption scheme:

```
{
        "p",
        "q",
        "g"
}
```

File 4 - encryptionParameters.json

Where *p* is the prime number representing the field $\mathbb{Z}_p$, *q* the large prime order of the cyclic subgroup and *g* a generator of the cyclic subgroup.

The information contained inside the **online** folder is explained in the following subsections.

2.1.1 The Authentication folder

The `authentication` folder contains two files with their corresponding signatures:

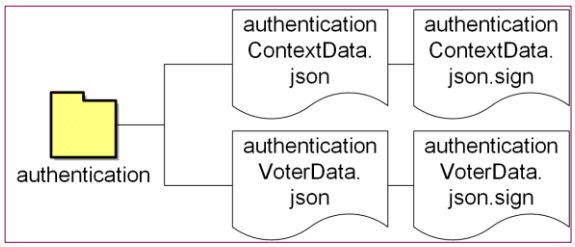


Figure 3 – Authentication folder

- The `authenticationContextData` file contains all the information required by the authentication context to run the authentication process, that is, to generate the challenge and the authentication token.

```
{
        "electionEventId",
        "authenticationTokenSignerKeystore",
        "authenticationTokenSignerPassword",
        "authenticationParams": {
                "challengeResExpTime",
                "authTokenExpTime",
                "challengeLength"
        }
}
```

File 5 - authenticationContextData.json

- The `authenticationVoterData` file contains the election certificates to be sent to the voting client during the authentication to check their validity.

```
{
        "authenticationTokenSignerCert",
        "electionEventId",
        "electionRootCA",
        "authoritiesCA",
        "credentialsCA ",
        "servicesCA ",
}
```

File 6 - authenticationVoterData.json

2.1.2 The Extended Authentication

The `extendedAuthentication` *folder* contains as many folders as Voting Card Sets with the required information to perform the first part of the authentication process. Each line of this file corresponds to one voter in the Voting Card Set.

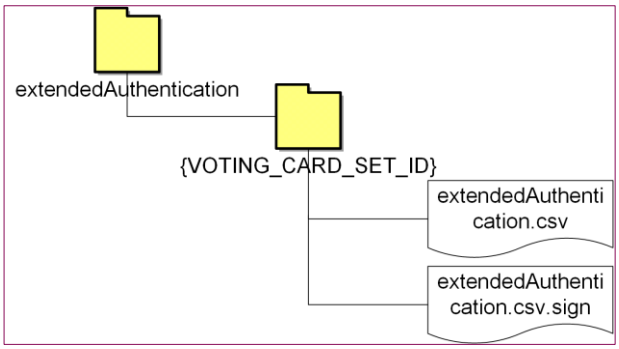


Figure 4 – ExtendedAuthentication folder

```
authID,extraAuhtParamBase64,encryptedSVK,eeid,saltBase64,credentialID
```

File 7 - extendedAuthentication.csv

2.1.3 The Electoral Authority

One electoral authority can be assigned to one or more Ballot Boxes, but one Ballot Box cannot have more than one electoral authority assigned.

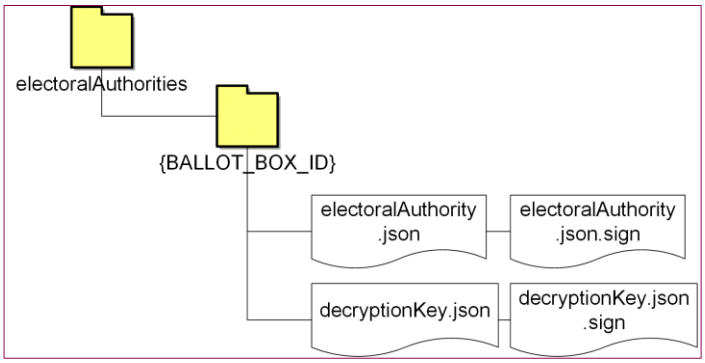


Figure 5 – ElectoralAuthorities folder

- The `electoralAuthority` file contains the election public key, that will be used to encrypt the votes:

```
{
        "id",
        "publicKey",
}
```

File 8 - electoralAuthority.json

- The value stored in the "`publicKey`" field is a JSON in base64 that has the following structure once decoded:

```
{
        "publicKey":{
                "zpSubgroup":{
                        "g":
                        "p":
                        "q":
                }
                "elements":{…}
}
```

Figure 6 - Public key JSON structure

- The `elements` field has as many elements as the number of components of the key.
- The `decryptionKey` file contains the Electoral Board public key:

```
{
        "electoralAuthorityId",
        "publicKey",
}
```

File 9 - decryptionKey.json

- The value stored in the `publicKey` field has the same structure as the public key mentioned above.

2.1.4 The Voting Workflow

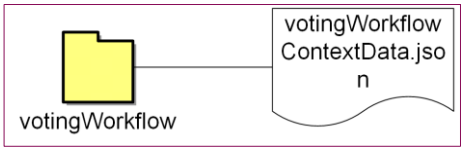


Figure 7 – VotingWorkflow folder

The `votingWorkflowContextData` file contains the election configuration to be used by the voting workflow context.

```
{
        "maxNumberOfAttempts",
}
```

File 10 - votingWorkflowContextData.json

2.1.5 The Voter Material

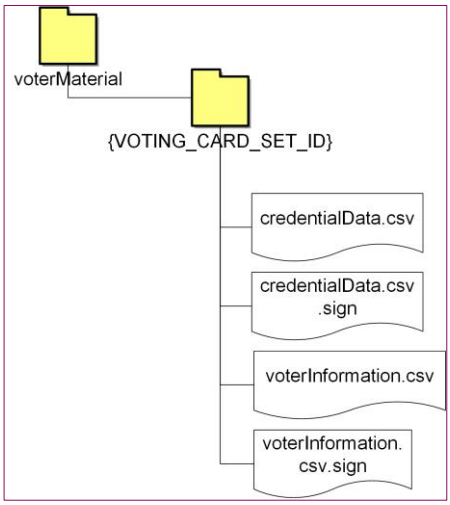


Figure 8 – VoterMaterial folder

For each voting card set, the `credentialData` file contains one row per voter belonging to that set with the following information:

```
VotingCardID, voterCredKeystoreBase64
```

File 11 - credentialData.csv

Each row of the `voterInformation` file contains the IDs of a specific voter that is part of the voting card set. The IDs are comma separated:

```
VotingCardID,BallotID,BallotBoxID,CredentialID,ElectionEventID,VotingCardSetID,VerificationCardId
,VerificationCardSetId
```

File 12 - voterInformation.csv

This information is uploaded to the voter material context.

2.1.6 The Vote Verification

The `voteVerification` folder is organized by Verification Card Sets and contains the following folders:

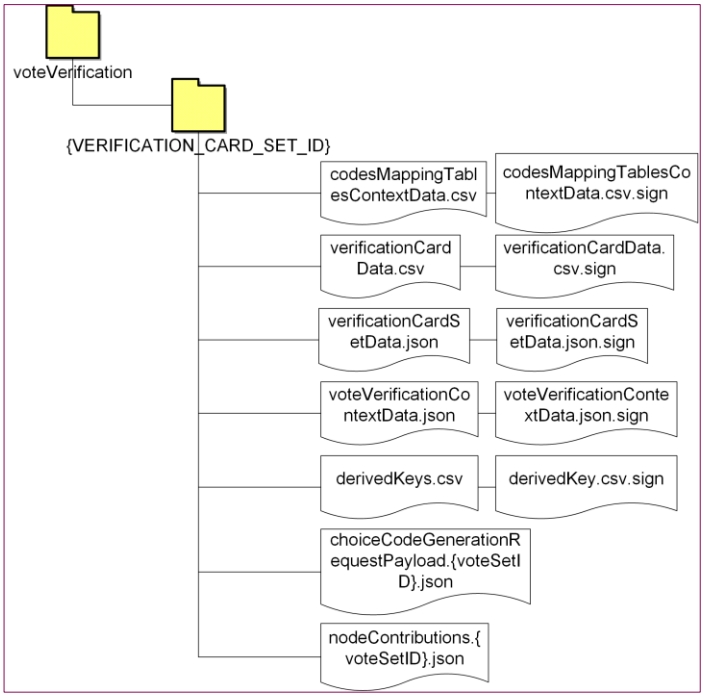


Figure 9 – VoteVerification folder

- Each line of the `codesMappingTablesContextData` file corresponds to one voter in the Verification Card Set and includes the verification card ID and the mapping table of that voter in base 64. The mapping table contains as many entries as Choice Return Codes and one additional entry for the Vote Cast Return Code.

```
VerificationCardId,MappingTableBase64
```

File 13 - codesMappingTablesContextData.csv

- Each line of the `verificationCardData` file corresponds to one voter in the Verification Card Set and contains the following data:

```
VerificationCardId,VerificationCardKeystoreBase64,VerificationCardPublicKe
yAndSignatureBase64, ElectionEventID, VerificationCardSetId
```

File 14 - verificationCardData.csv

- The `verificationCardSetData` file contains the information to be sent to the Client during the authentication phase. This information is common to all the voters belonging to the Verification Card Set.

```
{
        "electionEventId",
        "choicesCodesEncryptionPublicKeyBase64",
        "verificationCardIssuerCert",
        "verificationCardSetId",
        "voteCastCodeSignerCert"
}
```

File 15 - verificationCardSetData.json

- The `voteVerificationContextData` file contains all the information needed by the Vote Verification Context during the voting phase. The passwords included in this file are encrypted using the corresponding system key.

```
{
        "electionEventId",
        "encryptionParameters": {
                "p",
                "q",
                "g"
        },
        "verificationCardSetId",
        "electoralAuthorityId",
        "codesSecretKeyPassword",
        "codesSecretKeyKeyStore",
        "nonCombinedChoiceCodesEncryptionPublicKeys":{CCR1choiceCodesEncPKBase64;
        CCR2choiceCodesEncPKBase64;CCR3choiceCodesEncPKBse64;CCR4choiceCodesEncPK
        }
}
```

File 16 - voteVerificationContextData.json

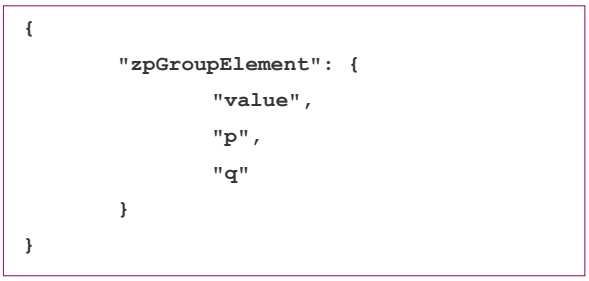
Each Choice Return Codes encryption public key (`CCR1choiceCodesEncPKBase64, CCR2choiceCodesEncPKBase64, CCR3choiceCodesEncPKBse64, CCR4choiceCodesEncPK`) is a JSON encoded in base64 that has the structure defined in Figure 6.

- The `derivedKeys` file contains the public keys generated in the Control Components (Voter Choice Return Codes generation public key and Voter Vote Cast Return Codes generation public key) corresponding to the derived voter's private keys. Each row of this file has the following information:

```
VerificationCardId,["CCR1VoterChoiceReturnCodeGenPKB64","CCR2VoterChoiceReturnCodeGenPKB64",
"CCR3VoterChoiceReturnCodeGenPKB64","CCR4VoterChoiceReturnCodeGenPKB64"],["CCR1VoterVoteCastReturnC
odeGenPKB64","CCR2VoterVoteCastReturnCodeGenPKB64",
"CCR3VoterVoteCastReturnCodeGenPKB64","CCR4VoterVoteCastReturnCodeGenPKB64"]
```

File 17 - derviedKeys.csv

Each Voter Choice Return Code Generation public key and each Voter Vote Cast Return Code Generation pubic key is a JSON encoded in base 64 that has the following structure:



```
{
        "zpGroupElement": {
                "value",
                "p",
                "q"
        }
}
```

Figure 10 - Voter Choice Return Code/Vote Cast Return Code generation public key JSON

- The `choiceCodeGenerationRequestPaylod.{voteSetId}` file contains the information sent to the Choice Return Codes Control Components during the configuration phase.

```
{
        "tenantId",
        "electionEventId",
        "verificationCardSetId",
        "chunkId",
        "choiceCodeGenerationInputList": [{
                "verificationCardId",
                "encryptedBallotCastingKey",
                "encryptedRepresentations"
        },{…}
        ],
        "signature":{
                "signatureContents",
                "certificateChain":[…]
        }
}
```

File 18 - choiceCodeGenerationRequestPayload_{voteSetId}.json

The field `choiceCodeGenerationInputList` contains as many elements as the number of Verification Card Ids in the Verification Card Set. The signature of this file is done using the Administration Board private key, and the corresponding certificate is included in the `signature.certificateChain` field.

- The `nodeContributions.{voteSetId}.json` file contains the information sent by the Control Components after computing the exponentiation of the encrypted prime numbers and the exponentiation of the encrypted ballot casting keys.

```
[{
        "correlationId",
        "requestId",
        "payload": {
                "tenantId",
                "electionEventId",
                "verificationCardSetId"
                "chunkId"
                "choiceCodeGenerationOutputList": [{
                        "verificationCardId",
                        "encryptedBallotCastingKey",
                        "computedBallotCastingKey",
                        "computedRepresentations",
                        "choiceCodesKeyCommitmentJson",
                        "ballotCastingKeyCommitmentJson"
                },{…}],
                "signature":{
                        "signatureContents",
                        "certificateChain":[…]
                }
        }
},{…}]
```

File 19 - nodeContributions_{voteSetId}.json

This JSON structure is repeated four times, one per Control Component. Inside the `choiceCodeGenerationOutputList` field there are as many elements as the number of Verification Card Ids in the Verification Card Set. The `computedBallotCastingKey` contains the exponentiated encrypted ballot casting key, the `computedRepresentations` contains the exponentiated encrypted prime numbers and the `choiceCodesKeyCommitmentJson` and `theballotCastingKeyCommitmentJson` contain the Voter Choice Return Codes generation public key and the Voter Vote Cast Return Code generation public key correspondingly. Notice that these keys must be equal to those stored in the File 17 file. Finally, the signature of the payload (`signature.signatureContents`) is computed using the Control Component signing private key and the corresponding certificate is stored in `signature.certificateChain`.

2.1.7 The Election Information

The `electionInformation` folder is organized by Ballots and Ballot Boxes, since one Ballot could belong to more than one Ballot Boxes.

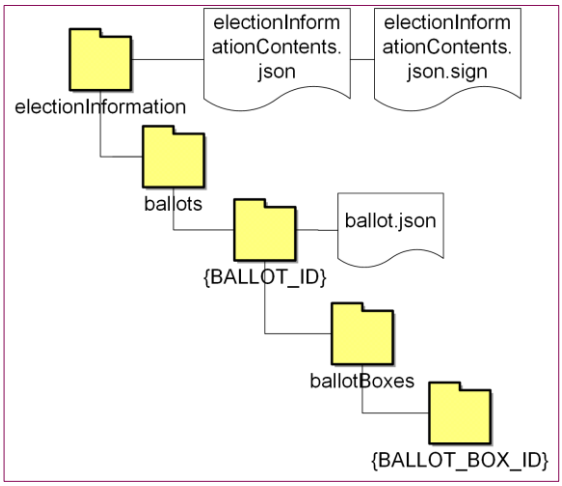


Figure 11 – ElectionInformation folder

- The `electionInformationContents` file contains the information used by the election information context to perform some validations during the voting phase.

```
{
        "electionEventId",
        "electionRootCA",
        "electionInformationParams":{
                "numVotesPerVotingCard",
                "numVotesPerAuthToken"
        },
        "authoritiesCA",
        "credentialsCA",
        "servicesCA"
}
```

File 20 - electionInformationContents.json

- The `ballot.json` contains the election information to be displayed to the voter, such as the questions and the possible answers. It also contains some rules to be enforced/checked on the voter's selections (for example, not selecting more than one answer).

```
        "id",
        "defaultTitle",
        "defaultDescription",
        "alias",
        "electionEvent":{
                "id"
        },
        "contests":[{
                "id",
                "defaultTitle",
                "alias",
                "defaultDescription":
                "electionEvent":{
                        "id"
                },
                "template",
                "fullBlank",
                "options":[{
                        "id",
                        "representation",
                        "attribute"
                }],
                "attributes":[{
                        "id",
                        "alias",
                        "correctness",
                        "related"
                }],
                "questions":[{
                        "id",
                        "max",
                        "min",
                        "accumulation",
                        "writeIn",
                        "blankAttribute",
                        "writeInAttribute",
                        "attribute",
                        "fusions":[]
                }],
                "encryptedCorrectnessRule",
                "decryptedCorrectnessRule",
        }],
        "status",
        "details",
        "synchronized",
        "ballotBoxes",
        "signedObject"
```

File 21 - ballot.json

Each folder corresponding to one Ballot Box contains the following information:

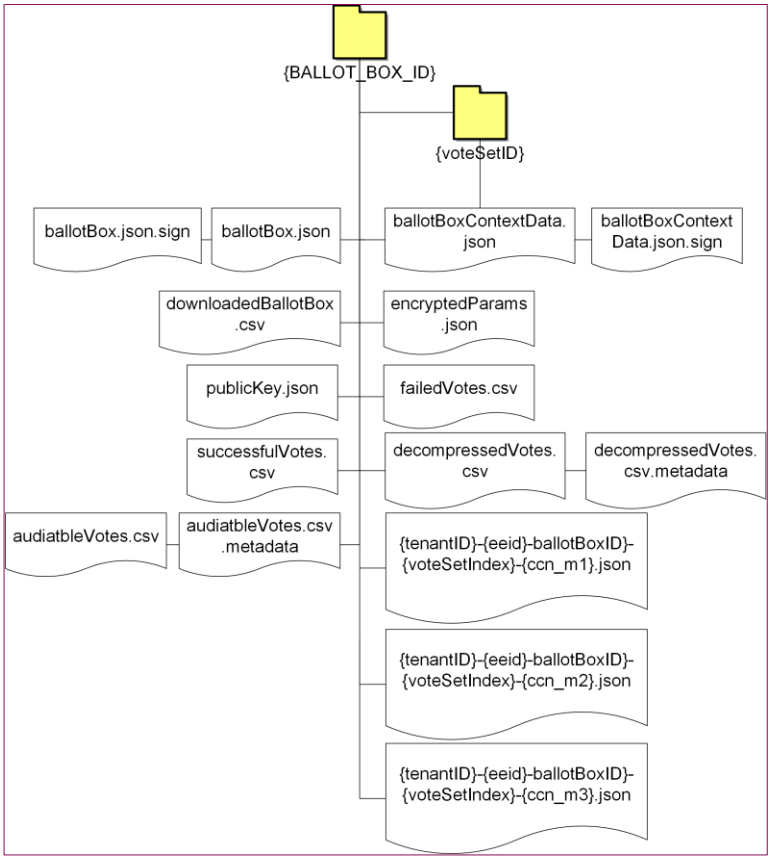


Figure 12 – Ballot Box folder

Most of the information needed to audit the counting process is stored in this folder.

- The `ballotBox` and `ballotBoxContextData` contain the election configuration related with one specific Ballot Box (for instance, the election dates), that will be uploaded to the election information context to be used during the voting phase.

```
{
        "id",
        "gracePeriod",
        "alias",
        "encryptionParameters": {
                "p",
                "q",
                "g"
        },
        "electoralAuthorityId",
        "writeInAlphabet",
        "confirmationRequired"
        "ballotBoxCert",
        "bid",
        "test"
        "startDate",
        "endDate",
        "eeid"
}
```


File 22 - ballotBox.json


```
{
        "id",
        "passwordKeystore",
        "keystore",
        "electionEvent": {
                "id",
        }
}
```


File 23 - ballotBoxContextData.json

The keystore password is encrypted with the corresponding system key.

- The `publicKey` file contains the election public key.
- The `downloadedBallotBox` file contains all the information stored in the Election Information Context database during the voting phase (see more details in section 2.2)
- The `successfulVotes` and `failedVotes` files are the output of the cleansing and they are stored for auditing purposes. The last row of each file contains their signature (see more details in section 2.3).
- The `decompressedVotes` file contains the decompressed decrypted voting options (see more details in section 2.4).
- The `auditableVotes` file contains the votes that have experimented some decryption error (see further details in section 2.5).

- The `{tenantID}-{eeid}-ballotBoxID}-{voteSetIndex}-{ccn_m1}.json`, `{tenantID}-{eeid}-ballotBoxID}-{voteSetIndex}-{ccn_m2}.json` and `{tenantID}-{eeid}-ballotBoxID}-{voteSetIndex}-{`*ccn_m3}.json* files, contain the output of the online Mixing Control Components. Their content is explained in more detail in section 2.4.

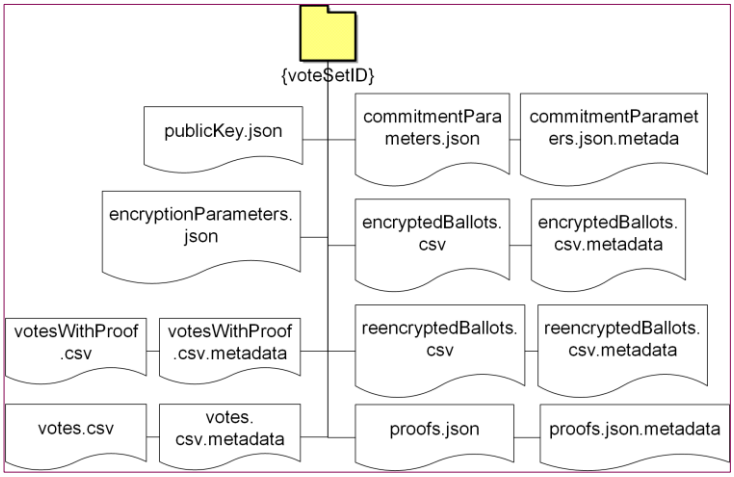


Figure 13 – VoteSetID folder

This folder contains the output of both the mixing and the decryption processes executed by the last Control Component, that will be explained in more detail in section 2.5.

2.2 Exported Ballot Box

The Exported Ballot Box is obtained from a dump of the Election Information Context database. When the Ballot Box is downloaded is signed using the corresponding Ballot Box private key. The Exported Ballot Box is stored in the following path:

`config/{election_event_id}/ONLINE/electionInformation/ballots/{ballot_id}/ballotBoxes/{ballot_box_id}/downloadedBallotBox.csv`

and its signature is stored in the last row of the CSV file. Each element stored in this file corresponds to one voter and contains the concatenation of the following information: the vote, the receipt and the authentication token separated by commas; and the Vote Cast Code, the Vote Cast Code signature, the Choice Codes Computations and the Vote Cast Code Computations sent by the Control Components, the Tenant ID, the Election Event ID, the Voting Card ID, the Ballot ID and the Ballot Box ID separated using pipes.

```
{"vote":{…}, "receipt":{…},"authenticationToken":{…}}|VoteCastCode|VoteCastCodeSignature|
ChoiceCodesComputations|VoteCastCodeComputations|TenantID|ElectionEventID|
VotingCardID|BallotID|BallotBoxID
```

File 24 - downloadedBallotBox.csv

The vote, the receipt and the authentication token have the following structure:

```
"vote":{
        "tenantId",
        "electionEventId",
        "ballotId",
        "ballotBoxId",
        "votingCardId",
        "encryptedOptions",
        "encryptedPartialChoiceCodes",
        "encryptedWriteIns",
        "correctnessIds"
        "verificationCardPublicKey",
        "verificationCardPKSignature",
        "signature"
        "certificate",
        "credentialId",
        "authenticationTokenSignature",
        "authenticationToken": "{…}",
        "schnorrProof":"{…}",
        "cipherTextExponentiations",
        "exponentiationProof":"{…}",
        "plaintextEqualityProof":"{…}",
        "verificationCardId",
        "verificationCardSetId"
},
```

```
"authenticationToken":{
        "id",
        "voterInformation":{
                "tenantId",
                "electionEventId",
                "votingCardId",
                "ballotId",
                "credentialId",
                "verificationCardId",
                "ballotBoxId",
                "votingCardSetId",
                "verificationCardSetId"
        },
        "timestamp",
        "signature"
}
```

```
"receipt":{
        "signature",
        "receipt",
}
```

The `schnorrProof`, `exponentiationProof` and `plaintextEqualityProof` contained in the vote, are represented as:

```
" proof ":"{
        "zkProof":{
                "q",
                "hash",
                "values":[value1,value2,…]
        },
}
```

Some of these fields have a particular representation that is detailed below:

FIELD	REPRESENTATION
verificationCardPublicKey	Base64
verificationCardPKSignature	Base64
Signature (vote)	Base64
Certificate	pem encoding
authenticationTokenSignature	Base64
Signature (receipt)	Base64
receipt	Base64
Id (authentication token)	Base64

Signature (authentication token)	Base64
proof.zkProof.q	Base64
proof.zkProof.hash	Base64
proof.zkProof.values	Base64

Table 1 - Representation fields

The following additional details are specifically relevant for the audit purposes of this document:

- The `encryptedOptions` field inside the vote contains two elements separated by the character ";" corresponding to the encryption of the product of prime numbers.

```
144039192105814016233257338645817818012342141387864310380279968635540856133436066063881724016789842073308109475324169057 32...;110186655328805695599739183010167483056285233496147467493622620364295559745459877174045911...
```

- The `encryptedPartialChoiceCodes` field inside the vote contains as many elements as voting options the voter can select plus 1, separated by the character ";".

- The `encryptedWriteIns` field inside the vote contains as many elements as write ins allowed by the ballot, separated by the character ";".

- The `cipherTextExponentiations` field inside the vote contains two elements separated by the character ";" related with the gamma element and the phi element that encrypts the product of prime numbers.

- The `ChoiceCodesComputations` are encoded using the class `java.util.zip.GZIPOutputStream`. In order to decode them, the following steps should be followed:

    - Decode base 64 and obtain a byte array

    - Unzip the result to text and obtain two JSON structures. The first one corresponds to the Control Components computations during the exponentiation of the partial Choice Return Codes and the second one contains the Control Components computations during the Choice Return Codes partial decryption.

```
[{
  "primeToComputedPrime":{partialCodeElement:exponentiatedPartialCodeElement},…
  "exponentiationProofJson": {
        "zkProof": {
              "hash",
              "values":[…],
              "q"
        }
  },
  "choiceCodesDerivedKeyJson"
  "signature": {
        "signatureContents",
        "certificateChain":[…]
  }
…]
```

File 25 - choiceReturnCodesComputationsJson

This JSON contains four times the data structure presented in the files above, one per Control Component.

- The `primeToComputedPrime` field contains as many pairs of `{partialCodeElement:exponentiatedPartialCodeElement}` as the number of elements of the encrypted partial choice codes.
- The `partialCodeElement` represents one element of this ciphertext, and the `exponentiatedPartialCodeElement` is the exponentiation computed by the Control Components using as basis de partial code element and as an exponent the voter's derived key, whose corresponding public key is stored in `choiceCodesDerivedKeyJson`.
- The exponentiation proof in `exponentiationProofJson` field is the proof computed by the Control Component to demonstrate that the exponentiation has been calculated using the right private key. Fields **`hash`**, **`values`** and **`q`** are encoded in base64.
- Finally, the `signatureContents` field contains the signature of the previous elements using the Control Component signing private key, and the `certificateChain` contains the certificate that must be used to validate the signature and also its certificate chain, more concretely, it contains 3 certificates: [Control Component Signing Certificate, Control Components CA, Platform Root CA].

Note: The public keys stored in the `choiceCodesDerivedKeyJson` must be equal to the Voter Choice Return Code Generation public keys stored in the File 17.

```
[{
 "decryptionContributionResult":[
        "zpGroupElement":{
                "value":,
                "p":,
                "q"
        }
        …
  ],
  "exponentiationProofJson": {
        "zkProof": {
                "hash",
                "values":[…],
                "q"
        }
  },
  "publicKeyJson"
  "signature": {
        "signatureContents",
        "certificateChain":[…]
  }
…]
```

File 26 - ChoiceReturnCodesDecryptionJson

This JSON contains four times the data structure presented in the files above, one per Control Component.

- The `decryptionContributionResult` field contains as many elements as the number of elements of the Control Component Choice Return Codes encryption key.
- The value in the `exponentiationProofJson` field is the proof computed by the Control Component to demonstrate that the exponentiation has been calculated using the right private key.
- Fields `hash`, `values` and `q` are encoded in base64.
- The `publicKeyJson` field contains the Control Component Choice Return Codes encryption public key.
- Finally, the `signatureContents` field contains the signature of the previous elements using the Control Component signing private key, and the `certificateChain` contains the certificate that must be used to validate the signature and also its certificate chain, more concretely, it contains 3 certificates: [Control Component Signing Certificate, Control Components CA, Platform Root CA].

Note: The Control Component Choice Return Code encryption public key elements from File 15 must be equal to the public key stored in the `publicKeyJson`.

- The `VoteCastCodeComputations` are encoded using the class `java.util.zip.GZIPOutputStream`. In order to decode them, the following steps should be followed:
    - Decode base 64 and obtain a byte array
    - Unzip the result to text and obtain the following JSON structure:

```
[{
  "primeToComputedPrime":{partialCodeElement:exponentiatedPartialCodeElement},
  "exponentiationProofJson": {
        "zkProof": {
                "hash",
                "values":[…],
                "q"
        }
  },
  "castCodeDerivedKeyJson"
  "signature": {
        "signatureContents",
        "certificateChain":[…]
  }
…]
```

File 27 - voteCastCodeComputationsJson

This JSON contains four times the data structure presented in the file above, one per Control Component.

- The `primeToComputedPrime` field contains one pairs of `{partialCodeElement:exponentiatedPartialCodeElement}` where `partialCodeElement` corresponds to the Confirmation Message and the `exponentiatedPartialCodeElement` is the exponentiation computed by the Control Components using as basis de partial code element and as an exponent the voter's derived key, whose corresponding public key is stored in `choiceCodesDerivedKeyJson`.
- The exponentiation proof in `exponentiationProofJson` field is the proof computed by the Control Component to demonstrate that the exponentiation has been calculated using the right private key. Fields **`hash`**, **`values`** and **`q`** are encoded in base64.
- Finally, the `signatureContents` field contains the signature of the previous elements using the Control Component signing private key, and the `certificateChain` contains the certificate that must be used to validate the signature and also its certificate chain, more concretely, it contains 3 certificates: [Control Component Signing Certificate, Control Components CA, Platform Root CA].

Note: The public keys stored in the `castCodeDerivedKeyJson` must be equal to the Voter Choice Return Code Generation public keys stored in the File 17.

2.3 Cleansed Ballot Box

The Cleansing process outputs are stored in `config/{election_event_id}/ONLINE/electionInformation/ballots/{ballot_id}/ballotBoxes/{ballot_box_id}`:

1) `successfulVotes.csv`: This file contains all the votes that have passed all the validations of the cleansing process. Each row of the CSV contains the following information corresponding to one vote: voting card ID, timestamp and receipt value. The receipt value is in Base64 representation. The last row of this file is the signature in base64.

```
VotingCardID;Timestamp;Receipt
```

File 28 - successfulVotes.csv

2) `failedVotes.csv`: This file contains the non-confirmed votes. Each row of the CSV file contains the following information corresponding to one failed vote: voting card ID, timestamp, error `(NOT_CONFIRMED)` and receipt value. The last row of this file is the signature in base64.

```
VotingCardID;Timestamp;Error;Receipt
```

File 29 - failedVotes.csv

Additionally, the cleansed votes are stored as part of the first Mixing node output, in the `previousVotes` field (see File 30 - CCM1, CCM2, CCM3 output in the next section).

2.4 Mixed and Decrypted Ballot Boxes in CCM_1, CCM_2 and CCM_3

The mixing and decryption processes executed in each online Control Component (CCM_1, CCM_2, CCM_3) output a JSON file with the mixed and decrypted Ballot Boxes, the corresponding proofs, and all the information needed to verify the operations.

This JSON files are stored in:

`config/{election_event_id}/ONLINE/electionInformation/ballots/{ballot_id}/ballotBoxes/{ballot_box_id}/{tenantID}-{eeid}-{ballotBoxID}-{voteSetIndex}-{ccn_m{1,2,3}}.json`

and contain the following information:

```
"voteEncryptionKey":{…},
"voteSetID":{…},
"votes":[…],
"electoralAuthorityId",
"encryptionParameters":{
        "g",
        "p",
        "q"
}
"decryptionProofs":[…],
"shuffledVotes":[…],
"shuffleProof":[…],
"commitmentParameters":[…],
"timestamp",
"signature":{
        "signatureContents".
        "certificateChain":[…]
},
"previousVotes":[…],
"previousVoteEncryptionKey":{…}
```

File 30 - CCM1, CCM2, CCM3 output

- The `voteEncryptionKey` contains the key with which the votes are encrypted after computing the partial decryption in the Control Component, or, in other words, is the `previousVoteEncryptionKey` without the contribution of the Control Component mixing public key.

  Note: In case of the first Control Component the `previousVoteEncryptionKey` will be the Election public key and in case of the third Control Component the `voteEncryptionKey` will be the Electoral Board public key. The structure of these fields is the following:

```
"previousVoteEncryptionKey":{
        "zpSubgroup":{
                "g",
                "p",
                "q"
        },
        "elements":[…]
}
```

```
"voteEncryptionKey":{
        "zpSubgroup":{
                "g",
                "p",
                "q"
        },
        "elements":[…]
}
```

File 31 - voteEncryptionKey structure file

- Before starting the mixing and decryption processes, in case the number of votes is too big to be processed at once, the Ballot Box is split into vote sets. The information in the `voteSetID` field is used to identify the vote set that has been mixed and decrypted.

```
"voteSetId":{
        "ballotBoxId":{
                "tenantId",
                "electionEventId",
                "id"
        },
        "index"
}
```

File 32 - voteSetId structure file

- The `previousVotes` field contains the ciphertexts that have been the input of the mixing executed in the current Control Component.

```
"previousVotes":[{
          "gamma",
         "phis":[…]
           },{…}
          }
```

File 33 - previousVotes structure file

- The `shuffledVotes` field contains the ciphertexts at the output of the mixing, that is, the ciphertexts in the `previousVotes` field mixed.

```
"shuffledVotes":[{
          "gamma",
         "phis":[…]
           },{…}
          }
```

File 34 - shuffledVotes structure file

- The `votes` field contains the ciphertexts at the output of the decryption, that is, the ciphertexts in the `shuffledVotes` field partially decrypted.

```
"votes":[{
          "gamma",
         "phis":[…]
           },{…}
          }
```

File 35 - votes structure file

- The `decryptionProofs` field contains as many decryption proofs as the number of votes that have been mixed and decrypted. The format of each proof is the following:

```
"zkProof":{
        "q",
        "hash",
        "values":[value1,value2,…]
}
```

File 36 - zkProof structure file

- The `shuffleProof` has the following structure:

initialMessage → $\overrightarrow{c_A}$
firstAnswer → $\overrightarrow{c_B}$

secondAnswer:

msgPA → represents the initial message of Product Argument

commitmentPublicB → c_b

iniHPA → Initial message of Hadamard Product Argument

commitmentPublicB → $\overrightarrow{c_B}$

ansHPA → Answer of Hadamard Product Argument

initial → Initial message of Zero Argument

commitmentPublicA0 → c_{A_0}

commitmentPublicBM → c_{B_m}

commitmentPublicD → $\overrightarrow{c_D}$

answer → Answer of Zero Argument

exponentsA → $\vec{a}$

exponentsB → $\vec{b}$

exponentR → r

exponentS → s

exponentT → t

iniSVA → represents the initial message of Single Value Product Argument

commitmentPublicD → c_d

commitmentPublicLowDelta → c_δ

commitmentPublicHighDelta → c_Δ

ansSVA → represents the answer of Single Value Product Argument.

exponentsTildeA → $\tilde{a}_1, \dots, \tilde{a}_n$

exponentsTildeB → $\tilde{b}_1, \dots, \tilde{b}_n$

exponentsTildeR → $\tilde{r}$

exponentsTildeS → $\tilde{s}$

iniMEBasic → initial message of multi-exponentiation argument

commitmentPublicA0 → c_{A_0}

commitmentPublicB → $\{c_{B_k}\}_{k=0}^{2m-1}$

ciphertextsE → $\{E_k\}_{k=0}^{2m-1}$

ansMEBasic → answer of multi-exponentiation argument

exponentsA → $\vec{a}$

exponentR → r

exponentB → b

exponentS → s

randomnessTau → τ

File 37 - Shuffle proof structure file

- The `commitmentParameters` contains a list of values that are: the encryption parameters (the first three values) and the generators used for the vector commitments:

$$p, q, g, H, G_1, \dots, G_n$$

File 38 - commitmentParameters.json

- The `signature` field contains the signature in base64 (`signature.contents`) and also the certificate chain (`signature.certificateChain`) needed to verify that signature in .PEM format: [Control Component Signing Certificate, Control Component CA Certificate].

2.5 Mixed and Decrypted Ballot Box in CCM_4

The output of the mixing and decryption processes executed in the last Control Component is stored in the following path:

`config/{election_event_id}/ONLINE/electionInformation/ballots/{ballot_id}/ballotBoxes/{ballot_box_id}/{voteSetID}`

- The `commitmentParameters.json` contains the information specified in File 38. The `commitmentParameters.json.metada` contains the file signature and the metadata used to compute it.
- Each line of the `encryptedBallots.csv` is one vote C with the following structure: $\alpha; \beta$, where $\alpha = g^r$ and $\beta = pk^r \cdot v$ according to the ElGamal encryption scheme. This is the input of the mixing process executed in the node and must be equal to the output of the previous online node (`votes` field in File 30). The `encryptedBallots.csv.metada` contains the file signature and the metadata used to compute it.
- Each line of the `reencryptedBallots.csv` file is one vote C' with the following structure: $\alpha; \beta$, where $\alpha = g^r$ and $\beta = pk^r \cdot v$ according to the ElGamal encryption scheme. This is the output of the mixing process executed in the node. The `reencryptedBallots.csv.metada` contains the file signature and the metadata used to compute it.
- The `publicKey.json` contains the key used by the mixing process to perform the re-encryption, that is, the Electoral Board public key.

```
{
        "publicKey":{
                "zpSubgroup":{
                        "g":
                        "p":
                        "q":
                }
                "elements":{…}
}
```

File 39 - publicKey.json

- The `votes.csv` file contains the output of the decryption process. Each line of this file corresponds to one decrypted vote and contains the product of primes numbers and in case write-ins are allowed, the encoded write-in text or the number 2 if the write-in has not been used. The `votes.csv.metada` contains the file signature and the metadata used to compute it.

$$\prod_{l=1}^{n} p_l\,;\{encodedWI\ or\ 2\},\dots,\{encodedWI\ or\ 2\}$$

File 40 - votes.csv

- The `votesWithProof.csv` file contains one row per encrypted vote with the following information in a JSON format: the encrypted vote, the decrypted vote and the decryption proof. The `votesWithProof.csv.metada` contains the file signature and the metadata used to compute it.

```
"[{"
        "gamma",
        "p",
        "q"
},
{"
        "phi1",
        "p",
        "q"
},
{…},
{"
        "phiM",
        "p",
        "q"
},

{"…"}]";
[decryptedValue];
"{" "zkProof":{
        "q",
        "hash",
        "values"
"}}"
```

File 41 - votesWithProof.csv

- The `proofs.json` file contains the mixing proof in the format specified in File 37 and the `proofs.json.metadata` contains its signature and the metadata used to compute it.

In addition to all the information mentioned previously, the decryption process outputs two more files that are stored in the Ballot Box folder (see Figure 12):

- The `decompressedVotes` file contains a list of decrypted and factorized voting options (factors $\{p_l\}_{l=1}^{n}$) and, in case write-ins have been used, it also includes the decoded text. Each line corresponds to one vote and individual factors are separated by ";".

$$p_1; p_2; \dots; p_n; p_{WI_1}; p_{WI_2}; \dots; p_{WI_1}\#writein_1; p_{WI_2}\#writein_2; \dots$$

File 42 - decompressedVotes.csv

If write-ins are allowed but they have not been used by the voter to vote for an option, the `decompressedVotes.csv.metada` contains the file signature and the metadata used to compute it.

- The `auditableVotes` file contains one line per vote that has experimented some decryption error.

```
timestamp;decryptionErrors;decryptedValue;factorization
```

File 43 - auditableVotes.csv

The errors can be one of the following:

```
RULE_VALIDATION
DUPLICATED_FACTOR
NON_FACTORIZABLE_REMAINDER
WRITE_IN_CONTENT_VIOLATION
```

- The `auditableVotes.csv.metada` contains the file signature and the information to validate it.

3 File signature verification

Files produces by the system components are signed to guarantee their security during the process. These signatures prevent file substitution and file content modifications.

3.1 Validate JSON files signature

There are three different ways to sign a JSON file and store its signature:

3.1.1 Use a metadata JSON file with the same `.json` file name plus `.metadata`

Given a JSON file, a `.json.metadata` file is created with the information used to compute the signature:

```
"version":"1.0"
"signed":[array of values used in the signature generation]
        {
            "field":[the name of the field included in the signature],
            "value": [the value of the field included in the signature]
        }
},
"alg":#{algorithm, padding and salt information}
"signature":#{base64 encoded string of the signature}
```

File 44 - metadata JSON file

Each element in the `"signed"` array contains the name and the value of the element included in the signature.

The simple approach to verify the file signature is to get the `signature` field from the metadata document and decode it into an array of bytes. Next, concatenate the original stream with the values of the fields in the **`signed`** array. Finally, verify the signature is valid against the stream.

More precisely, to verify the signature of the original resource you must use the metadata document and follow this process:

1) Read and convert to an array of bytes the **`signature`** field.
2) Read the **`signed`** array and check that if it is empty.
3) If the **`signed`** array is empty:
    - Validate that the signature verifies, using the bytes of the signature read in step 1, the original resource and the public key of the signer.
4) If the **`signed`** array is not empty, for each field:
    a) Concatenate the string representation of each field.
    b) Convert the string into an array of bytes.
    c) Concatenate the original resource (either as an array of bytes or java stream) with the byte array (of Java byte stream created from it) from step 4b.
5) Validate that the signature verifies, using the bytes of the signature read in step 1, the concatenated byte array (or Java stream) from step 4c and the algorithm specified in `alg`. **Note**: If no signature algorithm is provided, the default is "SHA256withRSAandMGF1" from Sun or Bouncy Castle provider.

```
public boolean verifySignature(final PublicKey publicKey, final InputStream metadataStream,
            final InputStream sourceStream) throws GeneralCryptoLibException {

try (JsonReader jsonReader = Json.createReader(metadataStream)) {
     final JsonObject metadataSignatureJson = jsonReader.readObject();
     final SignatureMetadata signatureMetadata =
SignatureMetadata.fromJsonObject(metadataSignatureJson);

     StringBuilder sb = new StringBuilder();
     signatureMetadata.getSignedFields().forEach((k, v) -> sb.append(v));
     String fieldsString = sb.toString();

     final byte[] bytes = fieldsString.getBytes(StandardCharsets.UTF_8);
     InputStream bs = new ByteArrayInputStream(bytes);
     InputStream seq = new SequenceInputStream(sourceStream, bs);

     byte[] signatureBytes =Base64.getDecoder().

     decode(metadataSignatureJson.getString(SignatureFieldsConstants.SIG_FIELD_SIGNATURE));

     return _verifier.verifySignature(signatureBytes, publicKey, seq);
    }
}
```

3.1.2 Use a file with the same `.JSON` file name plus `.sign`

Given a JSON file, a JWT[1] token is built from its contents, and then stored as a JSON object with one property, `.signature`, containing the JWT token. Each file is named as its source with `.sign` appended to it name. The signature verification involves parsing the JWT token back into a JSON object and comparing the fields with those in the original file.

[1] See IETF's RFC 7515

The following code is an example of how to verify these kinds of signatures:

```
public <T> T verify(PublicKey publicKey, String signedJSON, Class<T> clazz) {

    @SuppressWarnings("unchecked")
    Map<String, Object> claimMapRecovered =
            (Map<String, Object>) Jwts.parser().setSigningKey(publicKey).parse(signedJSON).getBody();

    final ObjectMapper mapper = new ObjectMapper();

    Object recoveredSignedObject = claimMapRecovered.get("objectToSign");
    return mapper.convertValue(recoveredSignedObject, clazz);
}
```

3.1.3 Store it in a field of the `.JSON` file

Given a JSON file whose signature is stored in one of its fields, the following mechanism can be used to verify the signature:

1) Obtain the signature from the . JSON file.
2) Remove the signature field from the . JSON file.
3) Verify the signature using the same methodology explained in the previous section.

3.2 Validate CSV files signature

There are three different ways to sign a `CSV` file and store its signature:

3.2.1 Use a metadata JSON file with the same `CSV` file name plus `.metadata`

The methodology used to verify the signature of a CSV file using the corresponding metadata file is the same that explained in section 3.1.1.

3.2.2 Use a file with the same `CSV` file name plus `.sign`

The methodology used to verify the signature of a CSV file using the corresponding .sign file is the following:

1) Read the signature from the `.csv.sign file`. This value is encoded in base64.
2) Decode the signature.
3) Given the public key (`java.security.PublicKey`), the signature (byte array) and the CSV file (InputStream), verify the signature using the algorithms implemented in `java.security.Signature.`

3.2.3 Store it in the last line of the `CSV` file

Given a CSV file whose signature is stored in the last line of it, the methodology proposed to verify the signature is the following:

1) Read the signature from the last line of the file. This value is encoded in base64.

2) Remove the signature from the file.
3) Decode the signature.
4) Given the public key (`java.security.PublicKey`), the signature (byte array) and the CSV file (InputStream), verify the signature using the algorithms implemented in `java.security.Signature`

4 Configuration validation

The following schema defines the certificate hierarchy of the system configuration and the table below the figure contains the certificate details.

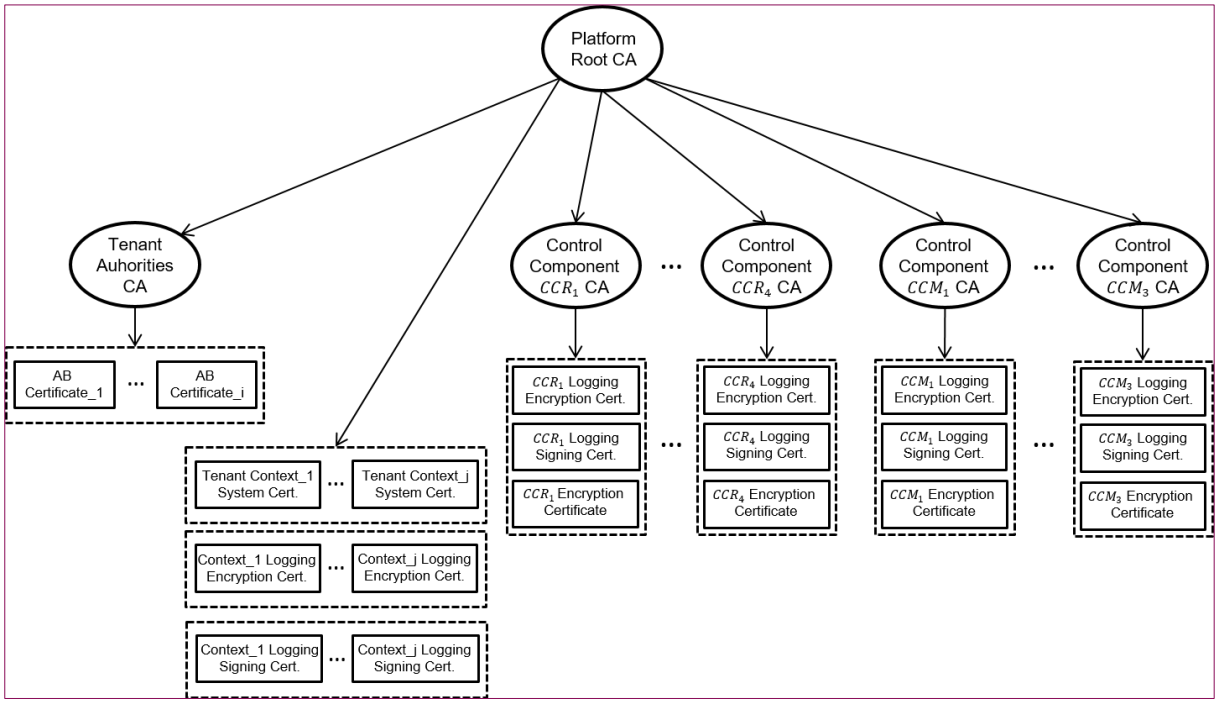


Figure 14 - System certificate hierarchy

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
Platform Root CA	${platformName} Root CA	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Tenant CA	Tenant ${tenantID} CA	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Control Component CCR_i CA	${CCRid} CA	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
CCR_i Logging Encryption Certificate	${CCRid} Log Encryption	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
CCR_i Logging Signing Certificate	${CCRid} Log Signer	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
CCR_i Encryption Certificate	${CCRid} ${platformID} Encryption	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Control Component CCM_j CA	${CCMid} CA	Organization	Online Voting	ES	CA	keyCertSign, cRLSign

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
CCM_i Logging Encryption Certificate	${CCMid} Log Encryption	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
CCM_i Logging Signing Certificate	${CCMid} Log Signer	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
CCM_i Encryption Certificate	${CCMid} ${platformID} Encryption	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Administration Board Certificate	AdministrationBoard ${adminBoardID}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Tenant Authentication Context System Certificate	AU ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Voting Workflow Context System Certificate	VW ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Vote Verification Context System Certificate	VV ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Voter Material Context System Certificate	VM ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Election Information Context System Certificate	EI ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Certificate Registry Context System Certificate	CR ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Tenant Extended Authentication Context System Certificate	EA ${tenantID} Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Authentication Context Logging	AU Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
Encryption Certificate						
Authentication Context Logging Signing Certificate	AU Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Voting Workflow Context Logging Encryption Certificate	VW Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Voting Workflow Context Logging Signing Certificate	VW Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Vote Verification Context Logging Encryption Certificate	VV Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Vote Verification Context Logging Signing Certificate	VV Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Voter Material Context Logging Encryption Certificate	VM Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Voter Material Context Logging Signing Certificate	VM Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Election Information Context Logging Encryption Certificate	EI Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Election Information Context Logging Signing Certificate	EI Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
Extended Authentication Context Logging Encryption Certificate	EI Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Extended Authentication Context Logging Signing Certificate	EI Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Certificate Registry Context Logging Encryption Certificate	EI Log Encryption	Organization	Online Voting	ES	Encryption	keyEncipherment, dataEncipherment
Certificate Registry Context Logging Signing Certificate	EI Log Signer	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation

Table 2 - System certificates details

In addition to the system certificates, the Online Voting system generates certificates related to a specific Election Event. The following two diagrams show the Election Event Certificate hierarchy and the Control Components Election Event Certificate hierarchy.

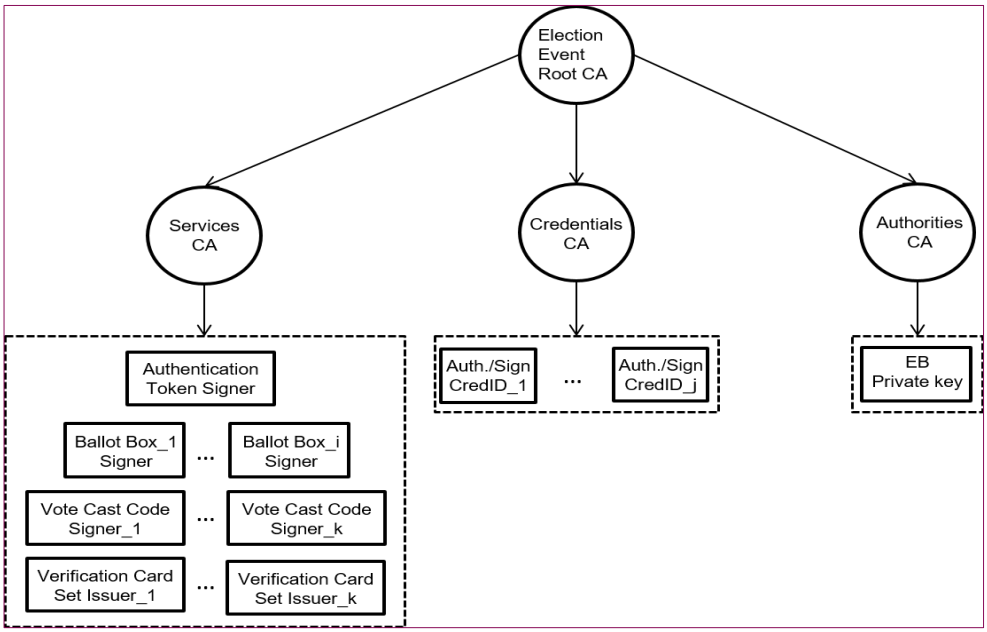


Figure 15 - Election Event certificate hierarchy

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
Election Event Root CA	Election Event Root CA ${eeid}	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Services CA	Services CA ${eeid}	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Authorities CA	Authorities CA ${eeid}	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Credentials CA	Credentials CA ${eeid}	Organization	Online Voting	ES	CA	keyCertSign, cRLSign
Authentication Token Signer	AuthTokenSigner ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Ballot Box Signer	BallotBox ${id}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Verification Card Set Issuer	VerificationCardIssuer ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Vote Cast Return Code Signer	VoteCastCodeSigner ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Credential ID signing	Sign ${cid}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation
Credential ID authentication	Auth ${cid}	Organization	Online Voting	ES	Sign	digitalSignature, nonrepudiation

Table 3 - Election Event certificates details

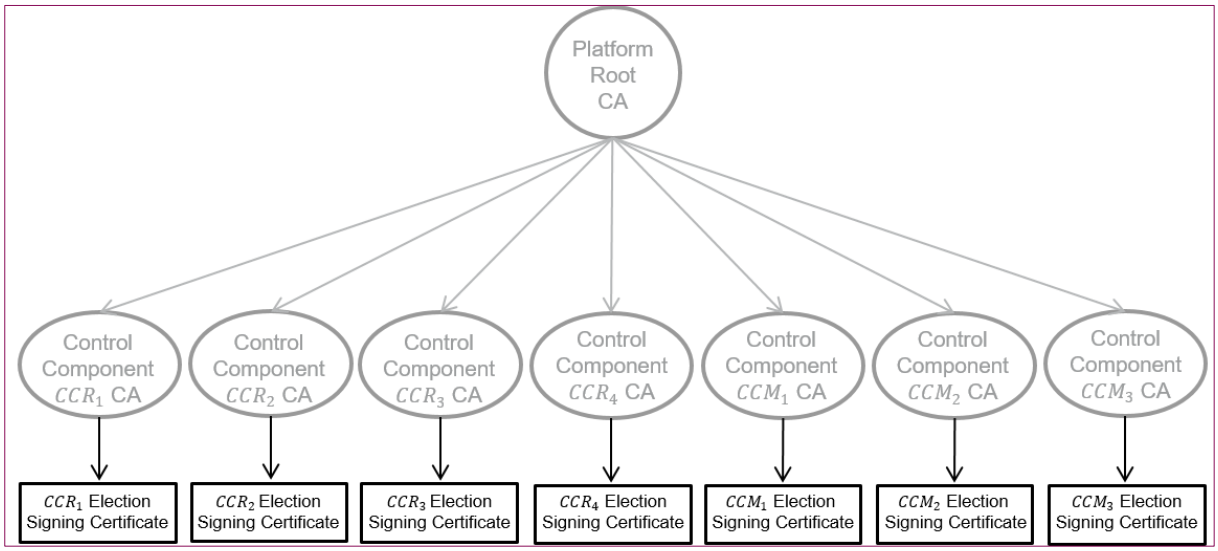


Figure 16 - Control Components Election Event certificate hierarchy

Certificate	Common name	Organization	Organizational Unit	Country	Key type	Key usage
CCR_1 **Signing Certificate**	${CCR1id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCR_2 **Signing Certificate**	${CCR2id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCR_3 **Signing Certificate**	${CCR3id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCR_4 **Signing Certificate**	${CCR4id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCM_1 **Signing Certificate**	${CCM1id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCM_2 **Signing Certificate**	${CCM2id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation
CCM_3 **Signing Certificate**	${CCM3id} ${eeid}	Organization	Online Voting	ES	Sign	digitalSignature, nonRepudiation

Table 4 - Control Components Election Event certificates details

4.1 Certificates validation

To validate the election certificates, the following strategy can be followed:

1) Obtain the certificate to be validated from the corresponding file:
    - Platform Root CA (see File 2).
    - Tenant CA (see File 3).
    - CCR_1 CA, CCR_2 CA, CCR_3 CA, CCR_4 CA, CCM_1 CA, CCM_2 CA, CCM_3 CA (Control Component data base or see Section 2.4).
    - CCR_1 signing certificate, CCR_2 signing certificate, CCR_3 signing certificate, CCR_4 signing certificate, CCM_1 signing certificate, CCM_2 signing certificate, CCM_3 signing certificate (Control Component data base or see Section 2.4).
    - Election Event Root CA, Authorities CA, Services CA, Credentials CA (see File 6 and File 20).
    - Authentication Token certificate (see File 6).
    - Administration Board certificate (see File 1).
    - For each Verification Card Set:
        - Vote Cast Return Code Certificate, Verification Card Set Issuer certificate (see File 15)
    - For each Ballot Box:
        - Ballot Box signing certificate (see File 22)

    All the certificates are in PEM format:

```
-----BEGIN CERTIFICATE-----
MIIDlDCCAnygAwIBAgIUDK2MyRFavMfrVbocJRewzVXOyr0wDQYJKoZIhvcNAQEL
BQAwXzEWMBQGA1UEAwwNVGVuYW50IDEwMCBDQTEWMBQGA1UECwwNT25saW5lIFZv
dGluZzEVMBMGA1UECgwMT3JnYW5pemF0aW9uMQkwBwYDVQQHDAAxCzAJBgNVBAYT
.....
-----END CERTIFICATE-----
```

File 45 - certificate in .PEM format

2) Obtain the parent certificate (see the certificates hierarchies at the beginning of the section).
3) Obtain the additional information (detailed in *Table 2*, *Table 3* and *Table 4*) needed to perform the validations.
4) Perform the certificate validations specified in section 11.1.9.

Voter's certificates are omitted in this section since their validation is explained in section 8.

4.2 Signatures validation

Configuration files presented in section 2.1 are signed using either the Administration Board Certificate or the corresponding Control Component Signing Certificate. Depending on which methodology has been used to sign the file, the signature is verified in a different way as explained in section3.

Validate the following file signatures using the methodology presented in section 3.1.2

- `authenticationContextData.json`
- `authenticationVoterData.json`
- `electionInformationContents.json`
- `ballotBox.json`
- `ballotBoxContextData.json`
- `electoralAuthority.json`
- `decryptionKey.json`
- `verificationCardData.json`
- `voteVerificationContextData.json`
- `verificationCardSetData.json`
- `commitmentParameters.json`

Validate the following file signatures using the methodology presented in section 3.1.3:

- `ballot.json`: before validating the signature of this file, remove the fields "status", "details" and "synchronized".

Validate the following file signatures using the methodology presented in section 3.2.2:

- `extendedAuthentication.csv`
- `credentialData.csv`
- `voterInformation.csv`
- `codesMappingTablesContextData.csv`
- `verificationCardData.csv`
- `derivedKeys.csv`

4.3 Control Components keys validation

The Control Components have their own keys to ensure both the privacy and the integrity of several processes executed during the configuration, voting and counting phases (the details of the generation and the usage of these keys are given in the protocol specifications document [1]):

- Control Component CA.
- Control Component signing key.
- Control Component Mixing key.
- Control Component Choice Return Code generation key.
- Control Components Choice Return Codes encryption public key.

The public part of these keys and the corresponding certificates can be found in the Control Components databases but additionally, the Control Component CA certificate is stored in the second element of the `signature.certificateChain` array in File 19, File 25, File 26, File 27 and File 30, and the Control Component Signing Certificate in the first element of `signature.certificateChain` array of the same files.

4.3.1 Choice Return Codes encryption key pair

The Choice Return Codes encryption key pair is generated among the Choice Return Codes Control Components (CCR) during the configuration phase. Each Control Component generates its own ElGamal Key pair, stores the private part of the key and sends the public part to the Secure Data Manager that multiplies all of them and finally obtains the Choice Return Codes encryption public key, included in File 16. In order to validate that the public key has been successfully consolidated:

1) Obtain the Choice Return Codes encryption public key encoded in Base64 from File 16.
2) Decode the key using the method defined in Appendix 11.3. Convert the decoded value to a BigInteger.
3) Obtain the Control Components Choice Return Codes encryption public key and its signature from File 16.
4) Decode the Control Components Choice Return Codes encryption public key using the method defined in in Appendix 11.3.
5) Obtain the Control Components signing certificates from the Control Component databases.
6) Validate the signature of the Control Components Choice Return Codes encryption public key concatenated with the Verification Card Set ID and the Election Event ID using the Control Components signing public key.
7) Convert the decoded values to BigIntegers.

8) Obtain the encryption parameters from File 4.
9) Multiply the Control Components Choice Return Codes encryption public keys. Do the operation modulo p.
10) Check that the result corresponds with the Choice Return Codes encryption public key.

4.3.2 Mixing key pair

The Election key is also generated among several entities: the mixing Control Components (CCM) and the Electoral Board. Each Control Component generates its own ElGamal Key pair, stores the private part of the key and send the public part to the Secure Data Manager. Once the Electoral Board is constituted, the Election key (included File 8) is computed by multiplying the Control Components Mixing public keys and the Electoral Board public key (File 9) to validate that the key has been successfully consolidated:

1) Obtain the Election public key encoded in Base64 File 8.
2) Decode the key using the method defined in Appendix 11.3. Convert the decoded value to a BigInteger.
3) Obtain the Electoral Board public key encoded in Base64 File 8.
4) Decode the key using the method defined in Appendix 11.3. Convert the decoded value to a BigInteger.
5) Obtain the Control Components Mixing public key and its signature from the Control Components database.
6) Decode the Control Components Mixing public key using the method defined in Appendix 11.3.
7) Obtain the Control Components signing certificates from the Control Components database.
8) Validate the signature of the Control Components Mixing public key concatenated with the Electoral Authority ID and the Election Event ID using the Control Components signing public key.
9) Convert the decoded values to BigIntegers.
10) Obtain the encryption parameters from File 4.
11) Multiply the Control Components Mixing public keys and the Electoral Board public key. Do the operation modulo p.
12) Check that the result corresponds with the Election public key.

5 Vote decompression validation

As is has been explained in section 2.5, the decryption process executed in the last Control Component will output the following files:

- `votes.csv` (see File 40)
- `votesWithProof.csv` (see File 41)
- `decompressedVotes.csv` and `decompressedVotes.csv.metadata` (see File 42)
- `auditableVotes.csv` and `auditableVotes.csv.metadata` (see File 43)

The following validations should be done in order to verify that the individual voting options obtained after the decompression were those decrypted.

1) Validate the signature of both the `decompressedVotes.csv` and `audtiableVotes.csv` files using the method explained in section 3.2.1 and the corresponding Ballot Box signing certificate (see File 22).
2) For each decrypted vote in the `votes.csv` file:
   - Take the first value, that is, the product of prime numbers $\prod_{l=1}^{n} p_l$ and factorize it using the information contained in the `ballot.json` (File 21). If the factorization is successful, check that the prime numbers obtained after it are included in the `decompressedVotes.csv` file. On the other hand, if any error happens, check that the decrypted vote and the corresponding error reason are included in the `auditableVotes.csv` file.
   - If write-ins are allowed the file should contain, in addition to the product of prime numbers, as many decrypted values as the number of write-ins. For each of these decrypted values: if the value is the number 2 it means that the write-in was not used by the voter to select an option and, consequently, there is no text to be decoded. If the value is different from 2, the number represents the encoded write-in and should be decoded in the following way:
     - Compute the square root modulo p.
     - Each character of the write-in is encoded using three numbers that represents its position in the following alphabet (included in the `ballot.json`):

```
#'(),-
./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ¢ŠšŽ
žŒœŸÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

     To avoid the usage of the number 2 in the encoding (remember that this number is used when the write-in is not selected), the process adds an offset of 2 to every character position. For example, the character “a” is in position 45, and it will be encoded as 047.
     In order to compute the decoding, first of all we need the result of the square root to be a multiple of 3. In case it is not, add as many 0s as necessary at the beginning of the value. Then, separate the resulting value in groups of 3

elements and for each one of them search in the alphabet for the corresponding character. Finally obtain a decoded value with the following format: $p_{WI_1}\#writein_1; p_{WI_2}\#writein_2; \ldots$

- If the decoding process is successful, check that the decoded values are included in the `decompressedVotes.csv` file and that the primers numbers $p_{WI_1}, p_{WI_2}, \ldots$ are part of the factorization computed in step 0). On the other hand, if any error happens, check that the decrypted vote and the corresponding error reason are included in the auditableVotes.csv file.

3) Run the `decryptedCorrectnessRule` inside the `ballot.json` (File 21) over the decrypted votes after being factorized. Check that the votes for which the rule has failed, are included in `auditableVotes.csv` (File 43).

6 Mixing and Decryption

In each of the Control Components the mixing and decryption processes are executed sequentially.

6.1 Validation of the CCM_4 output

Starting from the last Control Component, the following validations should be done in order to audit the mixing and decryption processes:

1) Validate the signature of the `votes.csv, votesWithProof.csv mencryptedBallots.csv,` the `reencryptedBallots.csv,` the `commitmentParameters.json` and the `proofs.json` using the Administration Board Certificate (File 1 - {adminBoardID}.pem) and the methodologies explained in sections 3.1.1 and 3.2.1.
2) For each tuple of (encryptedVote, decryptionProof, decryptedVote) in the votesWithProof file (File 41), call the **Decryption Proof verifier** with the following inputs:
   - Base elements: $[g\ , C_0]$
     - g is obtained from File 4
     - C_0 is the value stored in the `gamma` field.
   - Public input (group elements): $[pk_1, C'_1, pk_2, C'_2, \ldots, pk_m, C'_m]$
     - $[pk_1, pk_2, \ldots, pk_m]$ is the public key with which the votes are encrypted at the input of the decryption process, that is, the electoral board public key (File 9 and File 39).
     - $[C'_1, C'_2, \ldots, C'_m]$: These values are the result of dividing each phi element of the ciphertext (these values stored in the `phi` fields) by the corresponding value in the `decryptedValue` field.
   - Decryption proof
     - An exponent: `zkProof.hash` (c)

- An array of exponents: `zkProof.values` $(z_1; z_2; \ldots; z_m)$

These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers before using them.

- Additional data: "`DecryptionProof`"

3) Check that for each of the tuples in the votesWithProof.csv, the values stored in the `decryptedValue` field are also included in the votes.csv file.
4) Check that for each of the tuples in the `votesWithProof.csv`, the encrypted votes stored in the `gamma` and `phis` fields are also included in the `reencryptedBallots.csv`.
5) Call the **Mixing proof verifier** with the following inputs:
   - Input ciphertexts: list of ciphertexts stored in `encryptedBallots.csv` (see Section 2.5)
   - Output ciphertexts: list of ciphertexts stored in *reencryptedBallots.csv* (see Section 2.5)
   - Encryption parameters: values p, q and g in the `commitmentParameters.json` (File 38).
   - Commitment parameters: values $H, G_1, \ldots, G_n$ in the `commitmentParameters.json` (File 38).
   - Public key: Electoral Board public key elements stored in field `elements` of File 39.
   - Mixing proof: `proofs.json`. The information included in this file has the same structure than that presented in File 37.
6) Check that the ciphertexts in the `reencryptedBallots.csv` are the same ciphertexts included in the `votes` field of the previous Control Component output (File 30).

6.2 Validation of the CCM_1, CCM_2 and CCM_3 outputs

As it is explained in section 2.4, after the mixing and decryption are executed in the online Control Components a JSON file is generated with all the necessary information to validate that both processes have been executed as expected in each component.

1) Validate the signature of the output:
   - Concatenate the following information using the method *print* of the `java.io.PrintWriter` class: `voteSetId, votes, voteEncryptionKey, commitmentParameters, decryptionProofs, shuffledVotes, shuffleProof, timestamp, previousVotes` and `previousEncryptionKey`.
   - Obtain the signature from the field `signature.signatureContents` and decode it base64.
   - Obtain the Control Component Signing Certificate from the first element of the array stored in `signature.certificateChain` field.

- Verify the signature using the methods provided by `java.security.Signature`

2) For each element in the `votes` field search the corresponding decryption in the `decryptionProofs` field and the corresponding encrypted vote in the `shuffledVotes` field. Call the **Decryption Proof verifier** with the following inputs:
   - Base elements: $[g, C_0]$
     - g is obtained from File 4
     - C_0 is the value stored in the `votes.gamma` field.
   - Public input (group elements): $[pk_1, C'_1, pk_2, C'_2, \dots, pk_m, C'_m]$
     - $[pk_1, pk_2, \dots, pk_m]$ is the public key corresponding to the private key used to compute the partial decryption, that is, the Control Component Mixing public key.
     - $[C'_1, C'_2, \dots, C'_m]$: These values are the result of dividing each phi element of the ciphertext (these values stored in the `shuffledVotes.phi` fields) by the corresponding value in the `votes.gamma` fields.
   - Decryption proof
     - An exponent: `decryptionProofs.zkProof.hash` (c)
     - An array of exponents: `decryptionProofs.zkProof.values` $(z_1; z_2; \dots; z_m)$

     These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers before using them.
   - Additional data: "`DecryptionProof`"
3) Call the **Mixing proof verifier** with the following inputs:
   - Input ciphertexts: list of ciphertexts stored in the `previousVotes` field.
   - Output ciphertexts: list of ciphertexts stored in the shuffled field.
   - Encryption parameters: values g, p and q in the `encryptionParameters` field.
   - Commitment parameters: values $H, G_1, \dots, G_n$ in the `commitmentParameters` field. Notice that the first three values correspond to p, q and g.
   - Public key: public key stored in the `previousVoteEncryptionKey` field.
   - Mixing proof: proof stored in the `shuffleProof` field.

7 Cleansing validation

The cleansing determines, according to the system defined rules, which are the votes which are going to pass to the next phase (mixing and decryption). Then it removes all the information from the votes except for the encrypted voting options, which will be processed in further phases. Specifically, cleansing ensures that only one vote per Voting Card ID, and only if confirmed (if confirmation is required

during the voting phase), is considered in subsequent phases. The cleansing process outputs the following information:

- `successfulVotes.csv` (see File 28)
- `failedVotes.csv` (see File 29)
- Cleansed Ballot Box: as it is explained in section 2.3 the cleansed votes are stored as part of the first Mixing node output. The signature of these votes performed by the Voting Server is stored in the first Mixing Control Component logs, in an entry that has the following fields:
    - Log Event: Vote set signature successfully validated
    - Timestamp
    - Vote set signature (encoded Base64)
    - Control Component ID
    - Vote Encryption key (encoded Base64)
    - Votes (Cleansed votes encoded Base64)
    - Vote set ID

The following operations are proposed to audit the cleansing process, that is, to check that the output files of the process contain the correct information:

1) Verify the signatures of the successfulVotes.csv and failedVotes.csv files using the methodology proposed in section 3.2.3 and the Ballot Box signing certificate (File 22)
2) Verify the signature of the cleansed ballot box, that is, the signature of the first mixing node input, in the following way:
    - Decode base64 the values stored in the fields Vote set signature, Vote encryption key and Votes.
    - Concatenate the following information using the method *print* of the `java.io.PrintWriter` class: `voteSetId`, votes, `voteEncryptionKey` and timestamp
    - Obtain the Election Information signing certificate stored in the system keys folder (Figure 2).
    - Verify the signature using the methods provided by `java.security.Signature`
3) Look up the field `numVotesPerVotingCard` inside the `electionInformationContents.json` (File 20) to know how many votes are allowed per Voting Card ID. For each vote in the downloaded Ballot Box file, check if for its Voting Card ID, there are more votes in the Ballot Box that the number defined in `numVotesPerVotingCard`.

If this validation is not successful, the `failedVotes.csv` file should contain an entry with that Voting Card ID, the receipt corresponding to the vote and the error `DUPLICATE VOTE.`

4) For each vote in the downloaded Ballot Box (see File 24), check if the field corresponding to the Vote Cast Return Code is empty or not. If it is empty, this vote is not considered in the next phase and the `failedVotes.csv` file should contain an entry with that Voting Card ID, the receipt corresponding to the vote and the error `NOT CONFIRMED.`
5) Check that the downloaded Ballot Box contains exactly the votes in successfulVotes.csv and `failedVotes.csv`.

8 Ballot Box Validation

The following operations are proposed to validate the Ballot Box, that is, to verify the integrity and correctness of each Ballot Box and of the votes contained inside. The validations below should be done per each of the votes inside the downloaded Ballot Box, that have the structure explained in section 2.2.

8.1 Credential ID signing certificate validation

Validate the certificate using the following steps:

1) Obtain the Credential ID signing certificate (`vote.certificate`) that is in `.PEM` format (File 45).
2) Obtain the parent certificates: [Credentials CA, Election Event Root CA] (File 6 and File 20).
3) Obtain the additional information (detailed in *Table 2*, *Table 3* and *Table 4*) needed to perform the validations.
4) Perform the certificate validations specified in Appendix 11.1.9.

8.2 Signature validations

To verify the signatures, the `java.security.Signature` class is used. As it is explained in the documentation[2] three phases are required to perform the verification.

We explain in this section how to verify the signatures of the following elements:

- Vote Cast Return Code
- Authentication token
- Encrypted vote
- Receipt
- Verification Card Public Key

[2] https://docs.oracle.com/javase/7/docs/api/java/security/Signature.html

8.2.1 Vote Cast Return Code

1) Obtain the Vote Cast Return Code signature (`VoteCastCodeSignature`) and decode it using a Base64 decoder (Appendix 11.3). The result will be a byte array.
2) Obtain the Vote Cast Return Code (`VoteCastCode`) and the Verification Card ID (`voterInformation.verificationCardId`) and encode them as a byte array using the Charset UTF_8. These values are the signed information.
3) Obtain the Vote Cast Return Code certificate in .PEM format from File 15. The field is named `voteCastCodeSignerCert`. Recover the public key from it.
4) Verify the signature using the methods provided by `java.security.Signature`.

8.2.2 Authentication token

1) Obtain the authentication token signature from the `authenticationToken` field in the Ballot Box and decode it using a Base64 decoder (Appendix 11.3). The result will be a byte array.
2) Concatenate the following values (see an example on how to do it in Appendix 11.4), all inside the `authenticationToken` field.
    - Authentication token ID (`id`)
    - Tenant ID (`voterInformation.tenantId`)
    - Election Event ID (`voterInformation.electionEventId`)
    - Voting Card ID (`voterInformation.votingCardId`)
    - Ballot ID (`voterInformation.ballotId`)
    - Credential ID (`voterInformation.credentialId`)
    - Verification Card ID (`voterInformation.verificationCardId`)
    - Ballot Box ID (`voterInformation.ballotBoxId`)
    - Verification Card set ID (`voterInformation.verificationCardSetId`)
    - Voting Card Set ID (`voterInformation.votingCardSetId`)
3) Convert the result to a byte array. This is the signed information.
4) Obtain the Authentication token certificate in .PEM format from File 6. The field is named `authenticationTokenSignerCert`. Recover the public key from it.
5) Verify the signature using the methods provided by `java.security.Signature`

8.2.3 Encrypted vote

1) Obtain the Vote signature (`vote.signature`) and decode it using a Base64 decoder (Appendix 11.3). The result will be a byte array.
2) Concatenate in a string array (Appendix 11.4) the following values, all inside the `vote` field.
   - Encrypted options (`vote.encryptedOptions`)
   - Encrypted write-ins (`vote.encryptedWriteIns`)
   - Correctness IDs (`vote.correctnessIds`)
   - Verification card public key signature (`vote.verificationCardPKSignature`)
   - Authentication token signature (`vote.authenticationTokenSignature`)
   - Schnorr proof (`vote.schnorrProof`)
   - Voting Card ID (`vote.votingCardId`)
   - Election Event ID (`vote.electionEventId`)
3) Convert the result to a byte array. This is the signed information.
4) Obtain the Credential ID signing certificate in PEM format from File 45 (`vote.certificate`). Recover the public key from it.
5) Verify the signature using the methods provided by `java.security.Signature`

8.2.4 Receipt

1) Obtain the Receipt signature (`receipt.signature`) and decode it using a Base64 decoder (Appendix 11.3). The result will be a byte array.
2) Obtain the Receipt value (`receipt.receipt`) and encode it as a byte array (Appendix 11.3). This is the signed information.
3) Obtain the Ballot Box certificate in PEM format from File 22. The field is named `ballotBoxCert`. Recover the public key from it.
4) Verify the signature using the methods provided by `java.security.Signature`

8.2.5 Verification Card Public Key

1) Obtain the Verification Card Public key signature (`verificationCardPKSignature`) and decode it using a Base64 decoder (Appendix 11.3). The result will be a byte array.
2) Obtain the Verification Card Public key (`verificationCardPublicKey`) and decode it using a Base64 decoder (Appendix 11.3), the election event Id (`electionEventId`) and the verification card ID (`verificationCardId`) from the `vote` structure. This is the signed information.

3) Obtain the Verification Card Issuer certificate in .PEM format from File 15. Recover the public key from it.

4) Verify the signature using the methods provided by `java.security.Signature`.

8.3 Proofs validations

The proofs to be validated are:

- Schnorr proof
- Exponentiation proof validator
- Plaintext equality proof validator

8.3.1 Schnorr proof

The Schnorr proof is generated in the voting client after the voting options are encrypted, using the **Schnorr proof generator**. To verify the proof, call the **Schnorr proof verifier** with the following inputs:

- Base elements (group elements): `encryptionParameters.g` (File 4)
- Public input (group elements): first element of `encryptedOptions` field
- Schnorr proof: `vote.schnorrProof`
    - An exponent: `schnorrProof.zkProof.hash` (c)
    - An array of exponents: `schnorrProof.zkProof.values` (z)

  These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers.
- Additional data: "SchnorrProof:VoterID="+_voterID + "ElectionEventID="+ _electionEventID" The value of the voter ID is obtained from `vote.votingCardId` and the value of the election event ID from `vote.electionEventId`

8.3.2 Exponentiation proof validator

The Exponentiation proof is generated in the voting client after the voting options and the partial Choice Return Codes are encrypted, using the **Exponentiation proof generator.** To verify the proof, call the **Exponentiation proof verifier** with the following input:

- Base elements: [`encryptionParameters.g, vote.encryptedOptions`]
- Public input (exponentiated elements):[`vote.verificationCardPublicKey ,vote.cipherTextExponentiations`]
- Exponentiation proof: `vote.exponentiationProof`
    - An exponent: `exponentiationProof.zkProof.hash` (c)
    - An array of exponents: `exponentiationProof.zkProof.values` (z)

These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers.

- Additional data: "ExponentiationProof"

8.3.3 Plaintext equality proof validator

The Plaintext equality proof is generated in the voting client after the voting options and the partial Choice Return Codes are encrypted, using the **Plaintext equality proof generator.** To verify the proof, call the **Plaintext equality proof verifier** with the following input:

- Base elements (group elements): $[g, pk_{EL}, \frac{1}{\prod_{i=1}^{t} pk_{RC_i}}]$
  - Generator: `encryptionParameters.g` (File 4)
  - First element of the election public key: `publicKey` (File 8)
  - The compression of the Choice Return Codes encryption public key elements, inverted: `choicesCodesEncryptionPublicKeyBase64` (File 15).

  These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers
- Public input (group elements): $[C'_0, D_0, \frac{C'_1}{D'_1}]$
  - C'_0 is the first element of `vote.cipherTextExponentiations`.
  - C'_1 is the second element of `vote.cipherTextExponentiations`
  - D_0 is the first element of the `encryptedPartialChoiceCodes`.
  - D'_1 is the compression of the encrypted partial Choice Return Codes elements from the second to the last one.

  The `encryptedPartialChoiceCodes` values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers.
- Plaintext equality proof: `vote.plaintextEqualityProof`
  - An exponent: `plaintextEqualityProof.zkProof.hash` (c)
  - An array of exponents: `plaintextEqualityProof.zkProof.values` (z_1, z_2)

  These values should be decoded Base64 (Appendix 11.3) and converted to BigIntegers
- Additional data: "PlaintextEqualityProof".

8.4 Vote validations

The vote validations performed are

- Vote hash validation
- Vote format

- Vote matches signing certificate

8.4.1 Vote hash validation

1) Obtain from the vote structure the signature (`vote.signature`), the Verification Card public key signature (`vote.verificationCardPKSignature`), the Election Event ID (`vote.electionEventId`) and the Voting Card ID (`vote.votingCardId`). From the authentication token obtain the signature (`authenticationToken.signature`). Decode this values Base64 in case it is necessary (Appendix 11.3)
2) Concatenate all this information (Appendix 11.4)
3) We propose to compute the hash of the resulting byte array using the `java.security.MessageDigest`[3] class. The algorithm used is SHA_256.
4) Once the hash is obtained, encode the result using a Base64 encoding (Appendix 11.3) and construct a new string with it using the charset UTF_8.
5) Compare this value with the receipt value within the receipt structure (`receipt.receipt`)

8.4.2 Vote format

This validation is performed to check if the encrypted options, the encrypted write-ins and the encrypted partial Choice Return Codes have the expected number of elements and if these elements are group members.

1) Retrieve the encryption parameters from File 4.
2) Obtain the encrypted options from the vote structure: `vote.encryptedOptions`
   a) Split the encrypted options using as a separator the character ";" and check that there are 2 elements.
   b) For each one of the elements obtained check that the value is between 1 and p-1 and that the result of the operation: $value^q \ (mod\ q)$ is equal to 1.
3) Obtain the encrypted partial Choice Return Codes from the vote structure: `vote.encryptedPartialChoiceCodes`
   a) Obtain the correctness IDs from the vote structure: `vote.correctnessIds`
   b) Split the encrypted partial Choice Return Codes using as a separator the character ";" and check that there as many elements as the number of correctness IDs minus 1.
   c) For each one of the elements obtained check that the value is between 1 and p-1 and that the result of the operation: $value^q \ (mod\ q)$ is equal to 1.

[3] https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html

4) Obtain the encrypted write ins from the vote structure: `vote.encryptedWriteIns`

   a) For each one of the elements obtained check that the value is between 1 and p-1 and that the result of the operation: $value^q \ (mod\ q)$ is equal to 1.

   b) Given the information contained in the ballot regarding the maximum number of write-ins allowed per contest, check that number of encrypted write-ins is correct.

8.4.3 Vote matches signing certificate

Check that the credentialID within the vote structure is contained in the CredentialID signing certificate.

8.5 Codes Mapping Table validation

From the information stored in the Ballot Box and in the Control Components secure logs it can be validated that both the encrypted partial choice codes and the confirmation message have a valid entry in the Codes Mapping Table.

In order to perform these validations, the following steps can be followed for each Verification Card ID:

1) For each Control Component obtain from File 25 the values stored in the variable `exponentiatedPartialCodeElement`. The result will be four list of elements corresponding to the exponentiation computed by each Control Component over the encrypted partial choice codes:

$$CCR_1 : {E_2}^{k_{id}^1} = \left(g^{r'}, \left(pk_{CCR}^{(1)}\right)^{r'} \cdot pCC_1^{id}, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r'} \cdot pCC_\psi^{id}\right)^{k_{id}^1}$$

$$CCR_2 : {E_2}^{k_{id}^2} = \left(g^{r'}, \left(pk_{CCR}^{(1)}\right)^{r'} \cdot pCC_1^{id}, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r'} \cdot pCC_\psi^{id}\right)^{k_{id}^2}$$

$$CCR_3 : {E_2}^{k_{id}^3} = \left(g^{r'}, \left(pk_{CCR}^{(1)}\right)^{r'} \cdot pCC_1^{id}, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r'} \cdot pCC_\psi^{id}\right)^{k_{id}^3}$$

$$CCR_4 : {E_2}^{k_{id}^4} = \left(g^{r'}, \left(pk_{CCR}^{(1)}\right)^{r'} \cdot pCC_1^{id}, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r'} \cdot pCC_\psi^{id}\right)^{k_{id}^4}$$

   These values are also available in the field `#pc_comp` of the Control Component log whose corresponding log event is “Partial code successfully computed”.

2) Multiply the ciphertexts and obtain the encrypted pre-Choice Return Codes.

$$\prod_{j=1}^{4} {E_2}^{k_{id}^j} = \left(g^{r' \cdot \hat{k}}, \left(pk_{CCR}^{(1)}\right)^{r' \cdot \hat{k}} \cdot p_1^k, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r' \cdot \hat{k}} \cdot p_\psi^k\right)$$

   where $\hat{k} = \sum_{j=1}^{4} k_{id}^j$ and $k = k_{id} \cdot \hat{k}$.

3) From File 26 and for each Control Component obtain the values stored in the variable `decryptionContributionResult`. The result will be four lists of elements corresponding to the partial decryptions computed by each Control Component over the encrypted pre-Choice Return Codes. More concretely, each element of these lists is the gamma value of the

previous ciphertext ($g^{r' \cdot \hat{k}}$), exponentiated to the corresponding Control Component Choice Return Codes Encryption private key element:

$$CCR_1 : g^{r' \cdot sk^{(1)}_{CCR_1} \cdot \hat{k} \cdot}, \dots, g^{r' \cdot sk^{(\psi)}_{CCR_1} \cdot \hat{k}}$$

$$CCR_2 : g^{r' \cdot sk^{(1)}_{CCR_2} \cdot \hat{k} \cdot}, \dots, g^{r' \cdot sk^{(\psi)}_{CCR_2} \cdot \hat{k}}$$

$$CCR_3 : g^{r' \cdot sk^{(1)}_{CCR_3} \cdot \hat{k} \cdot}, \dots, g^{r' \cdot sk^{(\psi)}_{CCR_3} \cdot \hat{k}}$$

$$CCR_4 : g^{r' \cdot sk^{(1)}_{CCR_4} \cdot \hat{k} \cdot}, \dots, g^{r' \cdot sk^{(\psi)}_{CCR_4} \cdot \hat{k}}$$

These values are also available in the field `#pd_cc` of the Control Component log whose corresponding log event is “Partial decryption successfully computed”.

4) Multiply the partial decryptions and obtain a list of gammas exponentiated to the sum of Control Component Choice Return Codes Encryption private keys:

$$(g^{r' \cdot sk^{(1)}_{CCR} \cdot \hat{k}}, \dots, g^{r' \cdot sk^{(\psi)}_{CCR} \cdot \hat{k}})$$

5) Negate each element of the list computed in the previous step and multiply them for the corresponding element of the encrypted pre-Choice Return Codes. Obtain the pre-Choice Return Codes:

$$pC^{id}_1 = v^k_1 = g^{-r' \cdot sk^{(1)}_{CCR} \cdot \hat{k}} \cdot \left(pk^{(1)}_{CCR}\right)^{r' \cdot \hat{k}} \cdot v^k_1$$

$$\vdots$$

$$pC^{id}_\psi = v^k_\psi = g^{-r' \cdot sk^{(\psi)}_{CCR} \cdot \hat{k}} \cdot \left(pk^{(\psi)}_{CCR}\right)^{r' \cdot \hat{k}} \cdot v^k_\psi$$

6) Obtain the correctness IDs from the field `correctnessIds` in File 24. This is an array that contains as many arrays as the number of pre-Choice Return Codes.

7) For each of the pre-Choice Return Codes obtain the corresponding correctness IDs and concatenate them with the Verification Card ID (vc_{id}), the Election Event ID in the following way:

$$v^k_i || VC_{id} || EEID || \{attributes\}$$

Compute a hash of the result.

8) Check if the result of the hash computed in the previous step is a valid entry of the mapping table.

9) For each Control Component obtain from File 27 the values stored in the variable `exponentiatedPartialCodeElement`. The result will be four elements corresponding to the exponentiation computed by each Control Component over the hash of the Confirmation Message:

$$CCR_1 : (Hash(CM^{id})^2)^{kc^1_{id}}$$

$$CCR_2 : (Hash(CM^{id})^2)^{kc^2_{id}}$$

$$CCR_3 : (Hash(CM^{id})^2)^{kc^3_{id}}$$

$$CCR_4 : (Hash(CM^{id})^2)^{kc_{id}^4}$$

These values are also available in the field `#pc_comp` of the Control Component log whose corresponding log event is “Partial code successfully computed”.

10) Multiply the exponentiations and obtain the pre-vote cast return code.

$$\prod_{j=1}^{4}(Hash(CM^{id})^2)^{kc_{id}^j} = (Hash(CM^{id})^2)^{\widehat{kc}}$$

where $\widehat{kc} = \sum_{j=1}^{4} kc_{id}^j$.

11) Concatenate the pre-vote cast return code with the Verification Card ID (vc_{id}), the Election Event ID in the following way:

$$pVCC^{id} || VC_{id} || EEID$$

Compute a hash of the result.

12) Check if the result of the hash computed in the previous step is a valid entry of the mapping table.

8.6 Consistent IDs validation

Check that the following tuples of IDs are consistent:

- `(TenantID, vote.tenantId, vote.authenticationToken.voterInformation.tenantId, authenticationToken.voterInformation.tenantId)`
- `(ElectionEventId, vote.electionEventId, vote.authenticationToken.voterInformation.electionEventId, authenticationToken.voterInformation.electionEventId)`
- `(VotingCardId, vote.votingCardId, vote.authenticationToken.voterInformation.votingCardId, authenticationToken.voterInformation.votingCardId)`
- `(BallotId, vote.ballotId, vote.authenticationToken.voterInformation.ballotId, authenticationToken.voterInformation.ballotId)`
- `(BallotBoxId, vote.ballotBoxId, vote.authenticationToken.voterInformation.ballotBoxId, authenticationToken.voterInformation.ballotBoxId)`
- `(vote.credentialId, vote.authenticationToken.voterInformation.credentialId, authenticationToken.voterInformation.credentialId)`

8.7 Voter information validation

Check that the voter information inside the authentication token matches one of the entries in File 12.

8.8 Authentication token expiration time validation

To validate if the authentication token was expired when the vote was cast, the following operations are done:

1) Retrieve the start date and the end date from the Ballot Box (File 22) corresponding to that voter (`ballotBoxId`) . Both are date-time values with a time-zone in the ISO-8601 calendar system. They are represented using the class ZonedDateTime of the java API `java.time`. Once they are obtained, they are formatted using the ISO-like date-time formatter that formats or parses a date-time with the offset and zone if available with the time-zone offset for UTC (DateTimeFormatter.ISO_DATE_TIME.withZone(ZoneOffset.UTC))
2) Obtain the authentication token timestamp from the `authenticationToken` field (`authenticationToken.timestamp`) and convert it to a `ZonedDateTime` object.
3) Check that the authentication token timestamp is not before the start date neither after the election end date.

8.9 Control Components Validation

During the voting phase, Control Components generate several proofs to demonstrate that they have used the secret values they committed to during the configuration phase. As it has been explained in section 2.2, each vote in the downloaded Ballot Box contains these proofs, the signatures and also the information needed to verify the proofs in the fields `ChoiceCodesComputations` and `VoteCastCodeComputations` (see section 2.2).

For each vote and for each Control Component the following validations should be done:

- Validate encrypted partial Choice Return Codes exponentiation using the information from File 25:
    - Verify the exponentiation proof using the **Exponentiation proof verifier** with the following inputs:
        - Base elements: [generator (from File 4), list of `partialCodeElement`] (notice that this list of `partialCodeElement` must be equal to the ciphertext stored in `vote.encryptedPartialChoiceCodes`).
        - Public input (exponentiated elements): [`choiceCodesDerivedKeyJson`, `exponentiatedPartialCodeElement`]
        - Exponentiation proof: `exponentiationProofJson`
            - An exponent: `exponentiationProofJson.zkProof.hash`

- An array of exponents: `exponentiationProofJson.zkProof.values`

- Additional data: "ExponentiationProof"

- Concatenate the values in the `primeToComputedPrime`, the `exponentiationProofJson` and the `choiceCodesDerivedKeyJson` and verify the signature (`signature.signatureContents`) of the result using the corresponding Control Component signing public key stored in the first element of the array `signature.certificateChain`.

- Validate pre-Choice Return Codes partial decryption using the information from File 26:
  - Before validating the exponentiation, the proof does the following computations:
    - Multiply the Control Components contributions stored in the `partialCodeElement` field in File 25 an obtain a unique ciphertext corresponding to the encryption of the pre-Choice Return Codes.
    - Multiply the elements of the public key stored in the `publicKeyJson` and obtain a compressed public key.
    - Multiply the elements stored in the `decryptionContributionResult` field.
    - Additional data: "ExponentiationProof"
  - Verify the exponentiation proof using the **Exponentiation proof verifier** with the following inputs:
    - Base elements: [generator (from File 4), gamma element of the ciphertext corresponding to the encryption of the pre-Choice Return Codes]
    - Public input (exponentiated elements): [compressed public key, compressed decryption contribution]
    - Exponentiation proof: `exponentiationProofJson`
      - An exponent: `exponentiationProofJson.zkProof.hash`
      - An array of exponents: `exponentiationProofJson.zkProof.values`
    - Additional data: "ExponentiationProof"
  - Concatenate the values in the `decryptionContributionResult`, the `exponentiationProofJson` and the `publicKeyJson` and verify the signature (`signature.signatureContents`) of the result using the corresponding Control Component signing public key stored in the first element of the array `signature.certificateChain`.

- Validate confirmation message exponentiation using the information from File 27:
    - Verify the exponentiation proof using the **Exponentiation proof verifier** with the following inputs:
    - Base elements: [generator (from File 4), list of `partialCodeElement`].
    - Public input (exponentiated elements): [`castCodeDerivedKeyJson`, `exponentiatedPartialCodeElement`]
    - Exponentiation proof: `exponentiationProofJson`
        - An exponent: `exponentiationProofJson.zkProof.hash`
        - An array of exponents: `exponentiationProofJson.zkProof.values`
        - Additional data: "ExponentiationProof"
    - Concatenate the values in the `primeToComputedPrime`, the `exponentiationProofJson` and the `castCodeDerivedKeyJson` and verify the signature (`signature.signatureContents`) of the result using the corresponding Control Component signing public key stored in the first element of the array `signature.certificateChain`.

8.10 Secure Logs validation

Secure Logger produces immutable logs which are protected by means of cryptographic mechanisms, ensuring that nobody can manipulate the entries stored in the log without being detected.

The information stored in the log could be used to recognize any inconsistency in the votes cast and recorded in the Ballot Box. The validation of the immutable logs can be done either through features implemented in the Secure Logger or through a command line tool.

The clear text session key recorded in each checkpoint entry allows independent auditors to verify the HMAC chain from the previous block. The auditor can furthermore validate the integrity of the log by verifying the digital signatures.

All the components in the online voting system register their actions in secure logs and the secure logger keys of each one of them are stored in their corresponding databases.

In order to validate that all the votes and confirmations in the downloaded Ballot Box (File 24) have been processed by the Control Components and the voting server, access to the Election Information Context's logs and to the Control Components' logs and perform the following validations:

Using the Election Information Context's logs:

- For each vote in the downloaded ballot box check that there is an entry in the logs that contains the following information regarding the storage of the vote:

- Authentication token hash computed as it is shown below:

```
String hashBytesBase64 =
Base64.encodeBase64String(primitivesService.getHash(
   inputDataFormatterService.concatenate(id, tenantId, electionEventId,
   votingCardId, ballotId, credentialId, verificationCardId,
   ballotBoxId, verificationCardSetId, votingCardSetId, timestamp,
   signature)));
```

- Voting card ID
- Election event ID
- Ballot box ID
- Vote hash computed as shown below:

```
byte[] hashBytes = primitivesService.getHash(inputDataFormatterService.
      concatenate(vote.getFieldsAsStringArray()));
return Base64.getEncoder().encodeToString(hashBytes);
```

```
public String[] getFieldsAsStringArray() {
      String[] result = {encryptedOptions, encryptedWriteIns,
      correctnessIds, verificationCardPKSignature,
      authenticationTokenSignature, schnorrProof, votingCardId,
      electionEventId };
return result;
```

- Receipt value
- The message “Vote and receipt correctly stored”

- For each confirmed vote in the downloaded Ballot Box check that there is an entry in the logs that contains the following information regarding the storage of the confirmation of the vote:
  - The message “Vote Cast Return Code stored”
  - Voting Card ID
  - Election Event ID
- Check that all the votes registered in the logs exist in the Ballot Box, that is, check that votes have not been deleted from the Ballot Box after they have been stored or/and confirmed.

Using the Control Components’ logs:

- For each vote in the downloaded Ballot Box check that there is an entry in the logs with the following information after the Control Components have validated the vote proofs:
  - The message “Successful vote validation”
  - Voting Card ID
  - Election Event ID

- Hash of the Authentication Token
- Hash of the vote
- Encrypted options
- Encrypted Write Ins
- Correctness Ids
- Verification Card public key signature
- Schnorr proof
- Vote signature
- Control Component ID
- Validate the vote signature as it is explained in section 8.2.3

- For each vote in the downloaded Ballot Box check that there is an entry in the logs with the following information after the partial Choice Return Codes are computed:
  - The message “Partial code successfully computed”
  - Verification Card ID
  - Election Event ID
  - Encrypted partial Choice Return Codes
  - Exponentiated encrypted partial Choice Return Codes
  - Control Component ID
- For each vote in the downloaded Ballot Box check that there is an entry in the logs with following information after partial Choice Return Codes have been partially decrypted:
  - The message "Partial decryption successfully computed"
  - Verification Card ID
  - Gamma element of the exponentiated encrypted partial Choice Return Codes
  - Exponentiated gamma element
  - Control Component ID
- For each confirmed vote in the downloaded Ballot Box check that there is an entry in the logs with the following information after the confirmation has been validated:
  - The message “Confirmation Message logging”
  - Voting Card ID
  - Election Event ID

- Authentication Token signature
- Confirmation Message
- Confirmation Message signature
- Control Component ID

Validate the confirmation message signature concatenating the confirmation message, the authentication token signature, the voting card id and the election event id and using the Credential ID signing certificate.

- For each confirmed vote in the downloaded Ballot Box check that there is an entry in the logs with the following information after the confirmation message is exponentiated:
  - The message “Partial code successfully computed”
  - Verification Card ID
  - Election Event ID
  - Confirmation Message
  - Exponentiated Confirmation Message
  - Control Component ID
- Check that all the votes registered in the logs exist in the Ballot Box, that is, check that votes have not been deleted from the Ballot Box after they have been stored or/and confirmed.

9 Authentication Validation

To validate that the votes in the Ballot Box have been cast by authenticated voters, take the downloaded Ballot Box (File 24) and for each vote retrieve the authentication token:

```
"authenticationToken":{
        "id",
        "voterInformation":{
                "tenantId",
                "electionEventId",
                "votingCardId",
                "ballotId",
                "credentialId",
                "verificationCardId",
                "ballotBoxId",
                "votingCardSetId",
                "verificationCardSetId"
        },
        "timestamp",
        "signature"
}
```

Take also the Authentication Context Secure logs and look for a log that contains the following information:

- Voting Card ID and Election Event ID included in the voter information inside the authentication token.
- The authentication token ID.
- The message “Successful authentication token generation”.
- The hash of the authentication token in base64:
    - Concatenate the authentication token ID, the tenant ID, the election event ID, the voting card ID, the ballot ID, the credential ID, the verification card ID, the Ballot Box ID, the verification card set ID, the timestamp and the signature (all fields are in string format). In Appendix 11.4 there is an example on how to do this concatenation.
    - Compute the hash of the concatenated information.
    - Encode the result in base64 (Appendix 11.3).

If a register is found in the secure logs, the voter that cast the vote using this token was successfully authenticated. The validity of this token is checked in section 8.

10 References

[1] R. team, “SPC_sVOTE_RS_Protocol_Control_Components_V5.0,” 2018.

[2] S. Bayer and J. Groth, "Efficient Zero-Knowledge Argument for Correctness of a Shuffle," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Cambridge, UK, 2012.

[3] S. Bayer and J. Groth, “Efficient zero-knowledge argument for correctness of a shuffle,” in *Advances in Cryptology - EUROCRYPT 2012*, 2012.

[4] U. Maurer, "Unifying Zero-Knowledge Proofs of Knowledge," in *Progress in Cryptology - AFRICACRYPT 2009: Second International Conference on Cryptology in Africa*, Gammarth, Tunisia, 2009.

[5] *NIST. FIPS PUB 180-4. Seure Hash Standard (SHS),* August, 2015.

11 Appendix

11.1 Cryptographic primitives

11.1.1 Schnorr proof generator

Based in the Schnorr identification algorithm for proving knowledge of a secret exponent.

Input

- Base elements (group elements): $[g]$
- Exponents: $[r]$
- Public input (group elements): $[C_0] = [g^r]$
- Auxiliary data string "$Data$"
- The function PHI is defined by:
    - Number of inputs = number of exponents = 1
    - Number of outputs = number of elements of the public input array = 1
    - Base elements
    - Computation rules $[(1,1)]$

Operation

1) Commit step:
    a) Pick one random exponent a_1 since there is only one element in the array of exponents.
    b) Compute $(B_1) = PHI(a_1)$ in the following way:
        - The computation rules $[(1,1)]$ establish that to generate the output, the first element of the base elements array should be exponentiated to the first element of the exponents array: $B_1 = g^{a_1}$
2) Challenge step: compute the hash of the Phi function output, the public input and the string data $c = Hash(C_0 \| B_1 \| "Data")$.
3) Answer step: compute $z_1 = a_1 + c \cdot r$.

Output

- Schnorr Proof $(c; z_1)$.

11.1.2 Schnorr proof verifier

Based in the Schnorr identification algorithm for proving knowledge of a secret exponent.

Input

- Base elements (group elements): $[g]$

- Public input (group elements): $[C_0] = [g^r]$
- Schnorr proof: $(c; z_1)$
- Auxiliary data string "$Data$"
- The function PHI is defined by:
  - Number of inputs = number of exponents in the proofs (z_1) = 1
  - Number of outputs = number of elements of the public input array = 1
  - Base elements
  - Computation rules $[(1,1)]$

Operation

1) Check that $(c; z_1)$ are exponents of the mathematical group
2) Check that C_0 is a group element of the mathematical group
3) Compute $D_1 = PHI(z_1)$ in the following way:
   - The computation rules $[(1,1)]$ establish that to generate the output, the first element of the base elements array should be exponentiated to the first element of the exponents array: $D_1 = g^{z_1}$.
4) Compute $B_1 = (C_0)^{-c} \cdot D_1$. Notice that $B_1 = (g^r)^{-c} \cdot g^{z_1} = g^{-rc} \cdot g^{a_1+rc} = g^{-rc+a_1+rc} = g^{a_1}$
5) Compute $c' = Hash(C_0 \| B_1 \| "Data")$ and check that $c' = c$

Output

- OK / Not OK

11.1.3 Exponentiation proof generator

Input

- Base elements (group elements): $[\mathrm{h}_1, \mathrm{h}_2, \dots, \mathrm{h}_\mathrm{n}]$
- Exponents: $[\mathrm{sk}]$
- Public input (group elements): $[\mathrm{h}_1^{\mathrm{sk}}, \mathrm{h}_2^{\mathrm{sk}}, \dots, \mathrm{h}_\mathrm{n}^{\mathrm{sk}}]$
- Auxiliary data string "$Data$"
- The function PHI is defined by:
  - Number of inputs = number of exponents in the proofs (z_1) = 1
  - Number of outputs = number of elements of the public input array = n
  - Base elements

- Computation rules $[(1,1);(2,1);\dots;(n,1)]$

Operation

1) Commit step:

   a) Pick one random exponent a_1 since there is only one element in the array of exponents.

   b) Compute $(B_1, B_2, \dots, B_n) = PHI(a_1)$ in the following way:

   - The computation rules $[(1,1);(2,1);\dots;(n,1)]$ establish that in order to generate the i-th output, the i-th element of the base elements array should be exponentiated to the first element of the exponents array: $B_i = h_i^{a_1}$

2) Challenge step: compute the hash of the Phi function output and the public input $c = Hash(h_1^{sk} \| \dots \| h_n^{sk} \| B_1 \| \dots \| B_n \| "Data")$.

3) Answer step: compute $z_1 = a_1 + c \cdot sk$.

Output

- Schnorr Proof $(c; z_1)$.

11.1.4 Exponentiation proof verifier

Input

- Base elements (group elements): $[h_1, h_2, \dots, h_n]$
- Public input (group elements): $[h_1^{sk}, h_2^{sk}, \dots, h_n^{sk}]$
- Exponentiation proof: $(c; z_1)$.
- Auxiliary data string "$Data$"
- The function PHI is defined by:
  - Number of inputs = number of exponents in the proofs (z_1) = 1
  - Number of outputs = number of elements of the public input array = n
  - Base elements
  - Computation rules $[(1,1);(2,1);\dots;(n,1)]$

Operation

1) Check that $(c; z_1)$ are exponents of the mathematical group
2) Check that $[h_1^{sk}, h_2^{sk}, \dots, h_n^{sk}]$ are group elements of the mathematical group
3) Compute $(D_1, \dots D_n) = PHI(z_1)$ in the following way:

   - The computation rules $[(1,1);(2,1);\dots;(n,1)]$ establish that to generate the i-th output, the i-th element of the base elements array should be exponentiated to the first element of the exponents array: $D_i = h_i^{z_1}$

4) Compute $B_i = \left(h_i^{sk}\right)^{-c} \cdot D_i$. Notice that $B_i = (h_i^{sk})^{-c} \cdot h_i^{z_1} = h_i^{-sk \cdot c} \cdot h_i^{a_1 + c \cdot sk} = h_i^{-sk \cdot c + a_1 + sk \cdot c} = h_i^{a_1}$

5) Compute $c' = Hash(h_1^{sk} \| \ldots \| h_n^{sk} \| B_1 \| \ldots \| B_n \| "Data")$ and check that $c' = c$

Output

- OK / Not OK

11.1.5 Plaintext equality proof generator

Input

- Base elements (group elements): $[g, pk_{EB}, \frac{1}{\prod_{i=1}^{t} pk_{RC_i}}]$
- Exponents: $[r \cdot sk_{ID},\ l]$
- Public input (group elements): $[C_0', D_0, \frac{C_1'}{D_1'}]$
- Auxiliary data string "$Data$"
- The function PHI is defined by:
  - Number of inputs = number of exponents = 2
  - Number of outputs = number of elements of the public input array = 3
  - Base elements
  - Computation rules $[(1,1); (1,2); (2,1), (3,2)]$

Operation

1) Commit step:

a) Pick two random exponents a_1, a_2 since there are two elements in the exponent array.

b) Compute $(B_1, B_2, B_3) = PHI(a_1,\ a_2)$ in the following way:

- The computation rules $([(1,1); (1,2); (2,1), (3,2)])$ establish that to generate the first output the first element of the base elements array should be exponentiated to the first element of the exponents array. $B_1 = g^{a_1}$.
- For the second output the first element of the base elements array is exponentiated to the second element of the exponents array: $B_2 = g^{a_2}$
- The third output is computed in 2 steps:
  - The second element of the base elements array is exponentiated to the first element of the exponents array $B_3' = pk_{EB}^{a_1}$

- The third element of the base elements array is exponentiated to the second element of the exponents array $B_3'' = \left(\frac{1}{\prod_{i=1}^{t} pk_{RC_i}}\right)^{a_2}$

Compute the third output multiplying the partial outputs: $B_3 = B_3' \cdot B_3''$

2) Challenge step: compute the hash of the Phi function output, the public input and the string data $c = Hash(C_0' \| D_0 \| \frac{C_1'}{D_1'} \| \ B_1 \| B_2 \| B_3 \| "Data")$.

3) Answer step: compute $z_1 = a_1 + c \cdot (r \cdot sk_{ID})$ and $z_2 = a_1 + c \cdot l$.

Output

- Schnorr Proof $(c; z_1; z_2)$.

11.1.6 Plaintext equality proof verifier

Input

- Base elements (group elements): $[g, pk_{EL}, \frac{1}{\prod_{i=1}^{t} pk_{RC_i}}]$
- Public input (group elements): $[C_0', D_0, \frac{C_1'}{D_1'}]$
- Plaintext equality proof: $(c; z_1; z_2)$.
- Auxiliary data string "$Data$"
- The function PHI is defined by:
  - Number of inputs = number of exponents = 2
  - Number of outputs = number of elements of the public input array = 3
  - Base elements
  - Computation rules $[(1,1); (1,2); (2,1), (3,2)]$

Operation

1) Check that $c, z_1 \ and \ z_2$ are exponents of the mathematical group

2) Check that $[C_0', D_0, \frac{C_1'}{D_1'}]$ are group elements of the mathematical group

3) Compute $(D_1, D_2, D_3) = \ PHI(z_1, z_2)$ in the following way:

   - The computation rules ($[(1,1); (1,2); (2,1), (3,2)]$) establish that in order to generate the first output the first element of the base elements array should be exponentiated to the first element of the exponents array. $D_1 = \ g^{z_1}$.
   - For the second output the first element of the base elements array is exponentiated to the second element of the exponents array$D_2 = \ g^{z_2}$

- The third output is computed in 2 steps:
  - The second element of the base elements array is exponentiated to the first element of the exponents array $D'_3 = pk_{EL}^{z_1}$
  - The third element of the base elements array is exponentiated to the second element of the exponents array $D''_3 = \left(\frac{1}{\prod_{i=1}^{t} pk_{RC_i}}\right)^{z_2}$

Compute the third output multiplying the partial outputs: $D_3 = D'_3 \cdot D''_3$

4) Compute $B_1 = (C'_0)^{-c} \cdot D_1$, $B_2 = (D_0)^{-c} \cdot D_2$ and $B_3 = (\frac{C'_1}{D'_1})^{-c} \cdot D_3$.

5) Compute $c' = Hash(C'_0 \| D_0 \| \frac{C'_1}{D'_1} \| B_1 \| B_2 \| B_3 \| "Data")$ and check that $c' = c$

Output

- OK / Not OK

11.1.7 Decryption proof generator

Based in the Chaum-Pedersen protocol for proving equality of discrete logarithms.

Input

- Base elements: $[g, C_0]$
- Exponents: $(sk_1, \dots, sk_k)$.
- Public input (group elements): $[pk_1, C'_1, pk_2, C'_2, \dots, pk_k, C'_k]$
- Auxiliary data string "$Data$"
- Function PHI
  - Number of inputs = number of exponents = k
  - Number of outputs = number of elements of the public input array = $2k$
  - Base elements
  - Computation rules = $[(1,1); (2,1); (1,2); (2,2); (1,3); (2,3); \dots; (1,k); (2,k)]$

Operation

1) Commit step:

a) Pick as many exponents as the number of elements in the exponents array: $(a_1, \dots, a_k)$.

b) Compute $(B_1, B_2, \dots, B_{2k}) = PHI(a_1, \dots, a_k)$ in the following way:

- The computation rules ($[(1,1); (2,1); (1,2); (2,2); (1,3); (2,3); \dots; (1,k); (2,k)]$) establish that to compute the outputs each element in the base elements array should be

exponentiated to every element in the exponents array: $(B_1, B_2, \dots, B_{2k-1}, B_{2k}) = (g^{a_1}, {C_0}^{a_1}, \dots, g^{a_k}, {C_0}^{a_k})$.

2) Challenge step: compute the hash of the Phi function output, the public input and the string data $c = Hash(pk_1 \| C'_1 \| \dots \| pk_k \| C'_k \| B_1 \| \dots \| B_k \| "Data")$.

3) Answer step: compute $(z_1, \dots, z_k)$ as $z_j = a_j + c \cdot (sk_j)$.

Output

- Decryption Proof $(c; z_1; z_2; \dots; z_k)$.

11.1.8 Decryption Proof verifier

Based in the Chaum-Pedersen protocol for proving equality of discrete logarithms

Input

- Base elements: $[g\ , C_0]$
- Public input (group elements): $[pk_1, {C'}_1, pk_2, {C'}_2, \dots, pk_k, {C'}_k]$
- Decryption Proof $(c; z_1; z_2; \dots; z_k)$
- Auxiliary data string "$Data$"
- Function PHI
  - Number of inputs = number of exponents = k
  - Number of outputs = number of elements of the public input array $= 2k$
  - Base elements
  - Computation rules = $[(1,1); (2,1); (1,2); (2,2); (1,3); (2,3); \dots; (1,k); (2,k)]$

Operation

1) Check that $c, z_1, z_2, \dots, z_k$ are exponents of the mathematical group

2) Check that $pk_1, {C'}_1, pk_2, {C'}_2, \dots, pk_k, {C'}_k$ are group elements of the mathematical group.

3) Compute $(D_1, D_2, \dots, D_{2k}) = PHI(z_1, \dots, z_k)$ in the following way:

   - The computation rules ($[(1,1); (2,1); (1,2); (2,2); (1,3); (2,3); \dots; (1,k); (2,k)]$) establish that in order to compute the outputs each element in the base elements array should be exponentiated to every element in the exponents array: $(D_1, D_2, \dots, D_{2k-1}, D_{2k}) = (g^{z_1}, {C_0}^{z_1}, \dots, g^{z_k}, {C_0}^{z_k})$.

4) Compute $B_i = \left(pk_{\frac{i+1}{2}}\right)^{-c} \cdot D_i$ and $B_{i+1} = \left(C_{\frac{i+1}{2}}\right)^{-c} \cdot D_i$

5) Compute $c' = Hash(pk_1 \| C'_1 \| \dots \| pk_k \| C'_k \| B_1 \| \dots \| B_k \| "Data")$

Output

- OK / Not OK

11.1.9 X.509 Certificate Validation

The abstract class X.509Certificate inside the package `java.security.cert` provides a standard way to access all the attributes of a X.509 certificate.

Firstly, convert the certificates to be validated to X509Certificate object:

```
String credentialsCaPem =
       JsonUtils.getJsonObject(electionInformationContentJson).getString(CREDENTIALS_CA);

X509Certificate credentialsCa =
X509Certificate)PemUtils.certificateFromPem(credentialsCaPem);
```

Certificate validity

The abstract class X.509Certificate provides a method to check that a given date is within the certificate's validity period. The method is checkValidity() and receives as a parameter an object of type Date inside the java.util package.

To audit the validity of the certificate we propose that:

- If the audit date is within the certificate's validity period, use the current date as a parameter for the checkValidity() method.
- If the audit date is outside the certificate's validity period, use the election end date as a parameter for the checkValidity() method.

Subject DN validation

Compare the subject distinguished name of each of the certificates with the expected one. This value can be obtained using the method getSubjectX500Principal() from the X509Certificate. This method returns an object X500Principal and the method getName() returns the string representation of the subject distinguished name in rfc2253 format. The values to be compared are those contained inside the subject distinguished name:

- Subject Common Name (SCN)
- Organizational Unit (OU)
- Organization (O)
- Country (C)

To obtain these values, we propose to use the LDAP API as it is explained in the Annex 11.1.10.

The expected values for each one of the certificates are those specified in *Table 2*, *Table 3* and *Table 4*.

Issuer DN validation

Compare the subject distinguished name of the issuer certificate of each one of the certificates with the expected one. This value can be obtained using the method getIssuerX500Principal() from the X509Certificate. This method returns an object X500Principal and the method getName() returns the string representation of the subject distinguished name in rfc2253 format. The values to be compared are those contained inside the subject distinguished name:

- Subject Common Name (SCN)
- Organizational Unit (OU)
- Organization (O)
- Country (C)

To obtain these values, we propose using the LDAP API as it is explained in Annex 11.1.10.
The expected values for each one of the certificates are those specified in *Table 2*, *Table 3* and *Table 4*.

Key usage

Depending on the type of certificate, the key contained in it may have different purposes. The abstract class X.509Certificate inside the package java.security.cert provides a standard way to access all the attributes of an X.509 certificate. In order to validate the key type of the certificate we propose to use the method getKeyUsage() inside X.509Certificate class, that returns the KeyUsage extension of the certificate that is being validated, represented as an array of Booleans:

```
KeyUsage ::= BIT STRING {
        digitalSignature      (0),
        nonRepudiation        (1),
        keyEncipherment       (2),
        dataEncipherment      (3),
        keyAgreement          (4),
        keyCertSign           (5),
        cRLSign               (6),
        encipherOnly          (7),
        decipherOnly          (8)
}
```

The corresponding key usages of each certificate are detailed in *Table 2*, *Table 3* and *Table 4*.

Signature

To validate the signature of one certificate, first, get the public key from the issuer certificate using the method getPublicKey() defined in X509Certificate class. Then, use the method verify() in the same class to verify the signature of the certificate using the public key obtained in the first step.

11.1.10 Mixing proof generator

Based on Bayer and Groth proof of a shuffle [2].

Notice that as this is a specification for implementing the shuffle proof, matrices rows in the original paper are considered columns in this description and columns are considered rows.

Input

- m, n
- List of encrypted votes $\vec{C} = \{C_i\}_{i=1}^{N}$
- List of re-encrypted and permuted votes $\vec{C}' = \{C_i'\}_{i=1}^{N}$ (where $C_i' = C_{\pi(i)}\mathcal{E}_{pk}(1; \rho_i)$)
- List of re-encryption parameters $\vec{\rho} = \{\rho_i\}_{i=1}^{N}$
- Permutation $\vec{a} = \{a_1, \dots, a_N\} = \{\pi(1), \dots, \pi(N)\}$
- Mathematical group (p, q, g)
- Public key used to encrypt the votes: pk

Operation

1) Generate the commitment key ck:
   - Generate a group element using the **Group Element generation** primitive with input the mathematical group (p, q, g). The result is H.
   - Generate as many group elements as n using the **Group Element generation** primitive with input the mathematical group (p, q, g). The result is $G_1, \dots, G_n$.

   The commitment key is $ck = (G_1, \dots, G_n, H)$.
2) Given the permutation $\vec{a} = \{a_1, \dots, a_N\}$ arrange it in a matrix of m rows and n columns:

$$A = \begin{pmatrix} a_1 & \cdots & a_n \\ \vdots & \ddots & \vdots \\ a_{(m-1)\cdot n+1} & \cdots & a_N \end{pmatrix} = \begin{pmatrix} \overrightarrow{A_1} \\ \vdots \\ \overrightarrow{A_m} \end{pmatrix}$$

3) Commit to each row A_i (for $i = 1, \dots, m$) of the permutation matrix A using the **Commitment generation** primitive with the following inputs:
   - A random exponent $r_i \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed: $\overrightarrow{A_i}$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   Obtain the commitment $com_{ck}(\overrightarrow{A_i}; r_i)$.

   After committing to all the rows, define the vector of commitments as $\vec{c}_A = (com_{ck}(\overrightarrow{A_1}; r_1), \dots, com_{ck}(\overrightarrow{A_m}; r_m))$ and the vector of randomness as $\vec{r} = (r_1, \dots r_m)$.
4) Given the list of encrypted votes $\vec{C}$ arrange them in a matrix of m rows and n columns:

$$\begin{pmatrix} C_1 & \cdots & C_n \\ \vdots & \ddots & \vdots \\ C_{(m-1)\cdot n+1} & \cdots & C_N \end{pmatrix} = \begin{pmatrix} \vec{C}_1 \\ \vdots \\ \vec{C}_m \end{pmatrix}$$

5) Given the list of encrypted votes $\vec{C}'$ arrange them in a matrix of m rows and n columns:

$$\begin{pmatrix} C'_1 & \cdots & C'_n \\ \vdots & \ddots & \vdots \\ C'_{(m-1)\cdot n+1} & \cdots & C'_N \end{pmatrix} = \begin{pmatrix} \vec{C}'_1 \\ \vdots \\ \vec{C}'_m \end{pmatrix}$$

6) Concatenate the values of $\vec{C}, \vec{C}'$ and $\vec{c}_A$ in the following way:
   - For each element in $\vec{C}$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{C}'$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.

   Concatenate in a single value all the results obtained from the three steps above and compute a hash of the concatenation. Call the result $x = Hash(\vec{C}|\vec{C}'|\vec{c}_A)$.
7) Given x and the permutation $\vec{a} = \{a_1, \dots, a_N\}$, compute the exponentiation of x to each element on $\vec{a}$: $x^{a_i} \ mod \ p$. The result is $\vec{b} = \{b_1, \dots, b_N\} = \{x^{a_1}, \dots, x^{a_N}\}$.
8) Given $\vec{b}$ arrange it in a matrix of m rows and n columns:

$$B = \begin{pmatrix} b_1 & \cdots & b_n \\ \vdots & \ddots & \vdots \\ b_{(m-1)\cdot n+1} & \cdots & b_N \end{pmatrix} = \begin{pmatrix} \overrightarrow{B_1} \\ \vdots \\ \overrightarrow{B_m} \end{pmatrix}$$

9) Commit to each row $\overrightarrow{B_i}$ (for $i = 1, \dots, m$) using the **Commitment generation** primitive with the following inputs:
   - A random exponent $s_i \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed: $\overrightarrow{B_i}$
   - Commitment key $\boldsymbol{ck} = (G_1, \dots, G_n, H)$

   Obtain the commitment $com_{ck}(\overrightarrow{B_i}; s_i)$.

   After committing to all the rows, define the vector of commitment as $\vec{c}_B = (com_{ck}(\overrightarrow{B_1}; s_1), \dots, com_{ck}(\overrightarrow{B_m}; s_m))$ and the vector of randomness as $\vec{s} = (s_1, \dots s_m)$.
10) Concatenate the values of $\vec{C}, \vec{C}', \vec{c}_A$ and $\vec{c}_B$ in the following way:
   - For each element in $\vec{C}$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{C}'$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{c}_B$ convert it to a string and concatenate all of them in a single value.

   Concatenate in a single value all the results obtained from the three steps above and compute a hash of the concatenation. Call the result $y = Hash(\vec{C}|\vec{C}'|\vec{c}_A|\vec{c}_B)$.
11) Concatenate the values of $\vec{C}, \vec{C}', \vec{c}_A, \vec{c}_B$ and the number 1 in the following way:

- For each element in $\vec{C}$ convert it to a string and concatenate all of them in a single value.
- For each element in $\vec{C}'$ convert it to a string and concatenate all of them in a single value.
- For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
- For each element in $\vec{c}_B$ convert it to a string and concatenate all of them in a single value.

Concatenate in a single value all the results obtained from the three steps above and the number 1 and compute a hash of the concatenation. Call the result $z = Hash(\vec{C}|\vec{C}'|\vec{c}_A|\vec{c}_B|1)$.

12) For each element in $\vec{a}$ and $\vec{b}$ compute the following values:

$$\begin{aligned} d_1 &= y \cdot a_1 + b_1 \\ &\vdots \\ &\vdots \\ d_N &= y \cdot a_N + b_N \end{aligned}$$

The result is $\vec{d} = (d_1, \dots d_N)$.

13) For each element in $\vec{d}$ compute:

$$\begin{aligned} d_1 - z \\ \vdots \\ d_N - z \end{aligned}$$

$$\boldsymbol{b}^{PArg} = \prod_{i=1}^{N} (d_i - z)$$

arrange it in a matrix of m rows and n columns

$$A^{PArg} = \begin{pmatrix} d_1 - z & \cdots & d_n - z \\ \vdots & \ddots & \vdots \\ d_{(m-1)\cdot n+1} - z & \cdots & d_N - z \end{pmatrix} = \begin{pmatrix} \vec{A}_1^{PArg} \\ \vdots \\ \vec{A}_m^{PArg} \end{pmatrix}$$

14) For each element in $\vec{r}$ and $\vec{s}$ compute the following values:

$$\begin{aligned} t_1 &= y \cdot r_1 + s_1 \\ &\vdots \\ t_m &= y \cdot r_m + s_m \end{aligned}$$

The result is $\vec{t} = (t_1, \dots, t_m)$

15) Generate m commitments of the vector of length n: $(-z, \dots, -z)$ using the **Commitment generation** primitive with the following inputs:
    - Exponent: 0
    - List of elements to be committed: $(-z, \dots, -z)$
    - Commitment key $ck = (G_1, \dots, G_n, H)$

    Obtain the commitment $com_{ck}(-z, \dots -, z\ ; 0)$.

    After computing all the commitments, define the vector of commitments as $\vec{c}_{-z} = \left(com_{ck}(-z, \dots, -z; 0), \dots, com_{ck}(-z, \dots, -z; 0)\right)$

16) Compute the exponentiation of each element in $\vec{c}_A$ to the hash value y: $\vec{c}_A^{\,y}$

17) Compute the product of each element in $\vec{c}_A^{\,y}$ by the corresponding element in $\vec{c}_B$ and obtain $\vec{c}_D$: $\vec{c}_D = \vec{c}_A^{\,y} \cdot \vec{c}_B$.

18) Compute the product of each element in $\vec{c}_D$ by the corresponding element in $\vec{c}_{-z}$: $\vec{c}_A^{PArg} = \vec{c}_D \vec{c}_{-z}$

19) In case $m = 1$ the result of the operation above ($\vec{c}_A^{PArg}$), the matrix A^{PArg} and the vector $\vec{t}$, will have only one element and the protocol will not work (more precisely, the Hadamard product argument required by the Product argument). For this reason, the following modifications should be done:
    - Modify $\vec{c}_A^{PArg}$:
        - Generate a vector with n elements filled with 1: $(1, \dots, 1)$
        - Commit to the vector using the **Commitment generation** primitive with the following inputs:
            - Exponent 0
            - List of elements to be committed: $(1, \dots, 1)$
            - Commitment key $(G_1, \dots G_n, H)$
        - Reconstruct the vector $\vec{c}_A^{PArg}$ in the following way:
            - The first element of the vector is the value already computed: $\vec{c}_D \vec{c}_{-z}$
            - The second element of the vector is the commitment of the vector: $(1, \dots, 1)$ computed in the step above.
    - Modify A^{PArg}:
        - As the matrix A^{PArg} has only one row: $\vec{A}_1^{PArg}$, define a second row $\vec{A}_2^{PArg}$ containing n elements filled with 1: $(1, \dots, 1)$

$$A^{PArg} = \begin{pmatrix} d_1 - z & \cdots & d_n - z \\ 1 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} \vec{A}_1^{PArg} \\ \vec{A}_2^{PArg} \end{pmatrix}$$

    - Modify $\vec{t}$:
        - As the vector $\vec{t}$ has only element: t_1, define a second element $t_2 = 0$.

$$\vec{t} = (y \cdot r_1 + s_1, 0)$$

20) Use the **Product argument** with the following inputs:
    - $\vec{c}_A^{PArg}$
    - A^{PArg}
    - $\vec{t}$
    - $\boldsymbol{b}^{PArg} = \prod_{i=1}^{N}(d_i - z)$
    - The commitment key $ck = (G_1, \dots, G_N, H)$.

21) Given the list of re-encryption parameters $\vec{\rho}$ and the vector $\vec{b}$ compute $\rho = -\vec{\rho} \cdot \vec{b} = -\sum_{i=1}^{N} \rho_i \, b_i$

22) Define the vector $\vec{x} = (x_1, \dots, x_N) = (1, \dots, N)$.

23) Compute the exponentiation of each element in $\vec{C}$ to the corresponding element in $\vec{x}$:

$$\begin{matrix} C_1^{x_1} \\ \vdots \\ C_N^{x_N} \end{matrix}$$

Compute the product of these values $C^{MExpArg} = \prod_{i=1}^{N} C_i^{x^i}$.

24) Call the **Multi-exponentiation argument** with the following inputs:
   - $\vec{C}_1', \ldots, \vec{C}_m'$
   - $C^{MExpArg}$
   - $\vec{c}_B$
   - $\vec{B}_1, \ldots, \vec{B}_m$
   - $\vec{s}$
   - ρ
   - Mathematical group (p, q, g)
   - pk

Output

The output of the proof will consist of the following values:

- initialMessage → $\vec{c}_A$
- firstAnswer → $\vec{c}_B$
- secondAnswer:
   - msgPA → represents the initial message of Product Argument
      - commitmentPublicB → c_b
      - iniHPA → Initial message of Hadamard Product Argument
         - commitmentPublicB → $\vec{c}_B$
      - ansHPA → Answer of Hadamard Product Argument
         - initial → Initial message of Zero Argument
            - commitmentPublicA0 → c_{A_0}
            - commitmentPublicBM → c_{B_m}
            - commitmentPublicD → $\vec{c}_D$
         - answer → Answer of Zero Argument
            - exponentsA → $\vec{a}$
            - exponentsB → $\vec{b}$
            - exponentR → r
            - exponentS → s
            - exponentT → t
         - iniSVA → represents the initial message of Single Value Product Argument
            - commitmentPublicD → c_d
            - commitmentPublicLowDelta → c_δ
            - commitmentPublicHighDelta → c_Δ

- ansSVA → represents the answer of Single Value Product Argument.
  - exponentsTildeA → $\tilde{a}_1, \dots, \tilde{a}_n$
  - exponentsTildeB → $\tilde{b}_1, \dots, \tilde{b}_n$
  - exponentsTildeR → $\tilde{r}$
  - exponentsTildeS → $\tilde{s}$
- iniMEBasic → initial message of multi-exponentiation argument
  - commitmentPublicA0 → c_{A_0}
  - commitmentPublicB → $\{c_{B_k}\}_{k=0}^{2m-1}$
  - ciphertextsE → $\{E_k\}_{k=0}^{2m-1}$
- ansMEBasic → answer of multi-exponentiation argument
  - exponentsA → $\vec{a}$
  - exponentR → r
  - exponentB → b
  - exponentS → s
  - randomnessTau → τ

11.1.10.1 Multi-exponentiation argument

Input

- $\vec{C}_1, \dots, \vec{C}_m$
- C
- $\vec{c}_A$
- $A = (\vec{a}_1, \dots, \vec{a}_m)$
- $\vec{r} \in \mathbb{Z}_q^m$
- $\rho \in \mathbb{Z}_q$
- p, q, g (the encryption parameters)
- pk (public key used to encrypt the votes)

Operation

1) Generate n random elements between 1 and q-1 and construct the vector $\vec{a}_0$.
2) Generate the following random elements between 1 and q-1: $r_0 \leftarrow \mathbb{Z}_q$ and $b_0, s_0, \tau_0, \dots, b_{2m-1}, s_{2m-1}, \tau_{2m-1} \leftarrow \mathbb{Z}_q$
3) Set $b_m = 0,\ s_m = 0, \tau_m = \rho$
4) Commit to the vector $\vec{a}_0$ using the **Commitment generation** primitive with the following inputs:
   - The exponent r_0
   - List of elements to be committed: $\vec{a}_0$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_{A_0} = com_{ck}(\vec{a}_0; r_0)$

5) Commit to each element b_k $(k = 0, \dots, 2m-1)$ using the **Commitment generation** primitive with the following inputs:
   - The exponent s_k
   - List of elements to be committed: b_k (the list contains only one element)
   - Commitment key $ck = (G_1, H)$

   The result is the commitment $c_{B_k} = com_{ck}(b_k; s_k)$.

   After computing all the commitments, we will obtain the set of $2m$ commitments: $\{c_{B_k}\}_{k=0}^{2m-1}$

6) For each pair of elements (b_k, τ_k) for $k = 0, \dots, 2m-1$, call the **ElGamal encryption** primitive with the following inputs:
   - (p, q, g)
   - pk
   - g^{b_k}
   - τ_k

   The result is the encryption of g^{b_k} using τ_k as the randomness for encrypting: $\mathcal{E}_{pk}(g^{b_k}; \tau_k)$

   After computing all the encryptions, we will obtain $2m$ encryption: $\{\mathcal{E}_{pk}(g^{b_k}; \tau_k)\}_{k=0}^{2m-1}$

7) Given $\vec{a}_1, \dots, \vec{a}_m$ and $\vec{C}_1, \dots, \vec{C}_m$ compute, for each $k = 0, \dots, 2m-1$, the following products:

$$\prod_{\substack{i=0,j=0 \\ j=(k-m)+1}}^{m,m} \vec{C}_i^{\vec{a}_j}$$

   The exponentiation of a vector to another vector is defined as:

$$\vec{c}^{\vec{a}} = \prod_{j=1}^{n} c_j^{a_j}$$

8) Given the values generated in steps **6)** and **7),** compute the following $2m$ values:

$$E_0 = \mathcal{E}_{pk}(g^{b_0}; \tau_0) \prod_{\substack{i=0,j=0 \\ j=1-m}}^{m,m} \vec{C}_i^{\vec{a}_j}$$

$$E_1 = \mathcal{E}_{pk}(g^{b_1}; \tau_1) \prod_{\substack{i=0,j=0 \\ j=2-m}}^{m,m} \vec{C}_i^{\vec{a}_j}$$

$$\vdots$$

$$E_{2m-1} = \mathcal{E}_{pk}(g^{b_{2m-1}}; \tau_{2m-1}) \prod_{\substack{i=0,j=0 \\ j=m}}^{m,m} \vec{C}_i^{\vec{a}_j}$$

   The result is the set $E_k = \mathcal{E}_{pk}(g^{b_k}; \tau_k) \prod_{\substack{i=0,j=0 \\ j=(k-m)+1}}^{m,m} \vec{C}_i^{\vec{a}_j}$ for $k = 0, \dots, 2m-1$.

9) Concatenate the values of $\vec{C}, \vec{C}', \vec{c}_A, c_{A_0}, \{c_{B_k}\}_{k=0}^{2m-1}$ and $\{E_k\}_{k=0}^{2m-1}$ in the following way:
   - For each element in $\vec{C}$ convert it to a string and concatenate all of them in a single value.

- For each element in $\vec{C}'$ convert it to a string and concatenate all of them in a single value.
- For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
- Convert c_{A_0} to a string.
- For each element in $\{c_{B_k}\}_{k=0}^{2m-1}$ convert it to a string and concatenate all of them in a single value.
- For each element in $\{E_k\}_{k=0}^{2m-1}$ convert it to a string and concatenate all of them in a single value.

Concatenate in a single value all the results obtained from the three steps above and compute a hash of the concatenation. Call the result $x = Hash\left(\vec{C}\middle|\vec{C}'\middle|\vec{c}_A\middle|c_{A_0}\middle|\{c_{B_k}\}_{k=0}^{2m-1}\middle|\{E_k\}_{k=0}^{2m-1}\right)$.

10) Compute the following vector $\vec{x} = (x, x^2, \dots, x^m)$
11) Arrange the vectors $(\vec{a}_1, \dots, \vec{a}_m)$ in a matrix A having n rows and m columns:

$$A = (\vec{a}_1 \quad \cdots \quad \vec{a}_m) = \begin{pmatrix} a_1 & \cdots & a_{(m-1)\cdot n+1} \\ \vdots & \ddots & \vdots \\ a_n & \cdots & a_N \end{pmatrix}$$

12) Given $\vec{a}_0$, A and $\vec{x}$ compute $\vec{a} = \vec{a}_0 + A\vec{x}$, where the product of a matrix by a vector is done in the standard way.
13) Given $\vec{r}$, $\vec{x}$ and r_0 compute $r = r_0 + \vec{r} \cdot \vec{x}$, where $\vec{r} \cdot \vec{x}$ is the standard inner product $\vec{r} \cdot \vec{x} = \sum_{i=1}^{m} r_i x_i$.
14) Given b_0, $\{b_k\}_{k=0}^{2m-1}$ and $\vec{x}$, compute $b = b_0 + \sum_{k=1}^{2m-1} b_k x^k$.
15) Given s_0, $\{s_k\}_{k=0}^{2m-1}$ and $\vec{x}$, compute $s = s_0 + \sum_{k=1}^{2m-1} s_k x^k$.
16) Given τ_0, $\{\tau_k\}_{k=0}^{2m-1}$ and $\vec{x}$, compute $\tau = \tau_0 + \sum_{k=1}^{2m-1} \tau_k x^k$.

Output

- Output $\vec{a}, r, b, s, \tau$

11.1.10.2 Product argument

With this argument we can demonstrate that a set of committed elements have a particular product.

Input

- $\vec{c}_A = com_{ck}(A; \vec{r}) = (c_{A_1}, c_{A_2}, \dots, c_{A_m})$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1)
- $A = (\vec{a}_1, \dots, \vec{a}_m)$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1)
- $\vec{r} = (r_1, \dots, r_m)$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1)
- $b = \prod_{i=1}^{m} \prod_{j=1}^{n} a_{ij}$
- Commitment public key ck

Operation

1) Given the matrix A

$$A = \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_m \end{pmatrix} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

Compute the product of the elements of each column:

$$\prod_{i=1}^{m} a_{i1}, \prod_{i=1}^{m} a_{i2}, \ldots, \prod_{i=1}^{m} a_{in}$$

and define the vector $\vec{b} = (\prod_{i=1}^{m} a_{i1}, \prod_{i=1}^{m} a_{i2}, \ldots, \prod_{i=1}^{m} a_{in})$.

2) Commit to $\vec{b}$ using the **Commitment generation** primitive with the following inputs:
   - A random exponent $s \in \mathbb{Z}_q$ between 1 and q-1.
   - List of elements to be committed: $\vec{b}$
   - Commitment key $ck = (G_1, \ldots, G_n, H)$

   Obtain the commitment $c_b = com_{ck}(\prod_{j=1}^{m} a_{1j}, \ldots, \prod_{j=1}^{m} a_{nj}; s)$
3) Engage in a **Hadamard product argument** given as input $\vec{c}_A, c_b, \vec{b}, \vec{a}_1, \ldots, \vec{a}_m, \vec{r}, s$ (the name of the variables is the same here as in the Hadamard product argument)
4) Engage in a **Single value product argument** given as input:
   - $b^{SVPArg} = b$
   - $\vec{a}^{SVPArg} = \vec{b}$
   - $c_a^{SVPArg} = c_b$

11.1.10.3 Hadamard product argument

Input

- $\vec{c}_A = com_{ck}(A; \vec{r}) = (c_{A_1}, c_{A_2}, \ldots, c_{A_m})$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1.
- $c_b = com_{ck}(\vec{b}; s)$
- $\vec{b}$
- $\vec{a}_1, \ldots, \vec{a}_m$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1)
- $\vec{r} = (r_1, \ldots, r_m)$ (notice that in case $m = 1$ and according to what is explained in **step 19)** this vector will contain 2 elements instead of 1)
- s
- Commitment public key ck

Operation

1) If $m > 1$:
   - Given the matrix A

$$A = \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_m \end{pmatrix} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

Compute the vectors $\vec{b}_1, \dots, \vec{b}_m$ in the following way:

$$\vec{b}_1 = \vec{a}_1 = (a_{11}, a_{21}, \dots, a_{n1})$$

$$\vec{b}_2 = \vec{a}_1\vec{a}_2 = \left(\prod_{j=1}^{2} a_{1j}, \prod_{j=1}^{2} a_{2j}, \dots, \prod_{j=1}^{2} a_{nj} \right)$$

$$\vdots$$

$$\vec{b}_{m-1} = \vec{a}_1 \cdots \vec{a}_{m-1} = \left(\prod_{j=1}^{m-1} a_{1j}, \prod_{j=1}^{m-1} a_{2j}, \dots, \prod_{j=1}^{m-1} a_{nj} \right)$$

$$\vec{b}_m = \vec{a}_1 \cdots \vec{a}_m = \left(\prod_{j=1}^{m} a_{1j}, \prod_{j=1}^{m} a_{2j}, \dots, \prod_{j=1}^{m} a_{nj} \right) = \vec{b}$$

That is, each vector is computed as $\vec{b}_i = \prod_{z=1}^{i} \vec{a}_z$ where the multiplication of two vectors is the entry-wise product (given $\vec{x}$ and $\vec{y}$ of n element, the product $\vec{x}\vec{y}$ is defined as $\vec{x}\vec{y} = (x_1y_1, \dots, x_ny_n)$). Define the matrix B as:

$$B = \begin{pmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_m \end{pmatrix}$$

- Commit to the vectors $\vec{b}_2, \dots, \vec{b}_{m-1}$ (notice that for $\vec{b}_1$ and $\vec{b}_m$ we already have a commitment) using the **Commitment generation** primitive with the following inputs:
  - A random exponent $s_i \in \mathbb{Z}_q$ between 1 and q-1
  - List of elements to be committed: $\vec{b}_2$
  - Commitment key $ck = (G_1, \dots, G_n, H)$

  After committing to all the vectors, we will obtain the following commitments:

$$c_{B_2} = com_{ck}(\vec{b}_2; s_2)$$

$$\vdots$$

$$c_{B_{m-1}} = com_{ck}(\vec{b}_{m-1}; s_{m-1})$$

- Define the vector $\vec{s}$ as

$$\vec{s} = (s_1, s_2, \dots, s_{m-1}, s_m) = (r_1, s_2, \dots, s_{m-1}, s)$$

  Notice that the last value of the vector (s) is the randomness used in the commitment c_b and the first value of the vector (r_1) is the first randomness of vector $\vec{r}$ used in the commitment $\vec{c}_A$.

- Define the commitment to the matrix B as:

$$\vec{c}_B = com_{ck}(B; \vec{s}) = \left(com_{ck}(\vec{b}_1; s_1), c_{B_2}, \dots, c_{B_{m-1}}, com_{ck}(\vec{b}_m; s_m)\right)$$

  where, $com_{ck}(\vec{b}_1; s_1) = com_{ck}(\vec{a}_1; r_1)$ and $com_{ck}(\vec{b}_m; s_m) = c_b$.

2) If $m = 1$:

- Define $\vec{b}_1 = \vec{a}_1$

- Define $\vec{b}_2 = \vec{a}_1\vec{a}_2$
- The commitment to $\vec{b}_1$ is directly the commitment to $\vec{a}_1$: $com_{ck}(\vec{b}_1; s_1) = com_{ck}(\vec{a}_1; r_1)$
- The commitment to $\vec{b}_2$ is directly the commitment c_b: $com_{ck}(\vec{b}_2; s_m) = c_b$
- Define the vector $\vec{s}$ as $\vec{s} = (s_1, s_2) = (r_1, s)$, where r_1 is the first randomness of vector $\vec{r}$ used in the commitment $\vec{c}_A$ and s is the randomness used in the commitment c_b.
- Define the commitment to the matrix B as:

$$\vec{c}_B = com_{ck}(B; \vec{s}) = \left(com_{ck}(\vec{b}_1; s_1), (\vec{b}_2; s_2)\right) = (com_{ck}(\vec{a}_1; r_1), c_b)$$

3) Concatenate the values of $\vec{c}_A$, c_b and $\vec{c}_B$ in the following way:
   - For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
   - Convert c_b to a string.
   - For each element in $\vec{c}_B$ convert it to a string and concatenate all of them in a single value.

   Concatenate in a single value all the results obtained from the three steps above and compute a hash of the concatenation. Call the result $x = Hash(\vec{c}_A|c_b|\vec{c}_B)$
4) Concatenate the values of $\vec{c}_A$, c_b ,$\vec{c}_B$ and the number 1 in the following way:
   - For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
   - Convert c_b to a string.
   - For each element in $\vec{c}_B$ convert it to a string and concatenate all of them in a single value.

   Concatenate in a single value all the results obtained from the three steps above and the number 1 and compute a hash of the concatenation. Call the result $y = Hash(\vec{c}_A|c_b|\vec{c}_B|1)$.
5) If $m > 1$:
   - Given the vectors $\vec{b}_1, \dots, \vec{b}_{m-1}, \vec{b}_m$ and the hash x, compute the following values:

$$\vec{d}_1 = x^1\vec{b}_1 \bmod q$$
$$\vdots$$
$$\vec{d}_{m-1} = x^{m-1}\vec{b}_{m-1} \bmod q$$
$$\vec{d} = \sum_{i=1}^{m-1} x^i\vec{b}_{i+1} \bmod q$$

   - Given the vector $\vec{s}$ and the hash x, compute the following values:

$$t_1 = x^1 s_1 \bmod q$$
$$\vdots$$
$$t_{m-1} = x^{m-1} s_{m-1} \bmod q$$
$$t = \sum_{i=1}^{m-1} x^i s_{i+1} \bmod q$$

6) If $m = 1$:
   - Given the vectors $\vec{b}_1, \vec{b}_2$ and the hash x, compute the following values:

$$\vec{d}_1 = x^1 \vec{b}_1 \; mod \; q$$
$$\vec{d} = x^1 \vec{b}_2 \; mod \; q$$

- Given the vector $\vec{s}$ and the hash x, compute the following values:

$$t_1 = x^1 s_1 \; mod \; q$$
$$t = x^1 s_2 \; mod \; q$$

7) Commit to each vector $\vec{d}_i$ using the **Commitment generation** primitive with the following inputs:
   - The corresponding t_i
   - List of elements to be committed: $\vec{d}_i$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_{D_i} = com_{ck}(\vec{d}_i; t_i)$.

   After computing all the commitments, we will obtain $c_{D_1} \dots, c_{D_{m-1}}$ in case $m > 1$ and c_{D_1} in case $m = 1$.

8) Commit to the vector $\vec{d}$ using the **Commitment generation** primitive with the following inputs:
   - The corresponding t
   - List of elements to be committed: $\vec{d}$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_D = com_{ck}(\vec{d}; t)$.

9) Commit to the vector of n elements filled with the value -1 using the **Commitment generation** primitive with the following inputs:
   - The corresponding $(-1, \dots, -1)$
   - List of elements to be committed: 0
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_{-1} = com_{ck}(-\vec{1}; 0)$.

10) Engage in a **Zero argument** given as input:
   - $\vec{c}_A^{\,0Arg} = \left(c_{A_1}^{0Arg}, c_{A_2}^{0Arg}, \dots, c_{A_m}^{0Arg}\right) = (c_{-1}, c_{A_2}, \dots, c_{A_m})$ (if $m = 1$ this vector has only two elements (c_{-1}, c_{A_2})).
   - $\vec{c}_B^{\,0Arg} = \left(c_{B_0}^{0Arg}, c_{B_1}^{0Arg}, \dots, c_{B_{m-1}}^{0Arg}\right) = (c_D, c_{D_1} \dots, c_{D_{m-1}})$ (if $m = 1$ this vector has only two elements (c_D, c_{D_1})).
   - $A^{0Arg} = \left(\vec{a}_1^{\,0Arg}, \vec{a}_2^{\,0Arg}, \dots, \vec{a}_m^{\,0Arg}\right) = (-\vec{1}, \vec{a}_2, \dots, \vec{a}_m)$ and $\vec{r}^{\,0Arg} = \left(r_1^{0Arg}, \dots, r_m^{0Arg}\right) = (0, r_2, \dots, r_m)$ (if $m = 1$ these vectors have only two elements $(-\vec{1}, \vec{a}_2)$,$(0, r_2)$)
   - $B^{0Arg} = (\vec{b}_0^{\,0Arg}, \vec{b}_1^{\,0Arg}, \dots, \vec{b}_{m-1}^{\,0Arg}) = (\vec{d}, \vec{d}_1, \dots, \vec{d}_{m-1})$ and $\vec{s}^{\,0Arg} = \left(s_0^{0Arg}, \dots, s_{m-1}^{0Arg}\right) = (t, t_1, t_2, \dots, t_{m-1})$ (if $m = 1$ these vectors have only two elements $(\vec{d}, \vec{d}_1)$,(t, t_1))

11.1.10.4 Zero argument

Input

- $\vec{c}_A = com_{ck}(A; \vec{r})$
- $\vec{c}_B = com_{ck}(B; \vec{s})$
- $(\vec{a}_1, \dots, \vec{a}_m)$ (the rows of matrix A. Notice that in case $m = 1$ this vector contains 2 elements according to that explained in **step 19)**).
- $\vec{r} = (r_1, \dots, r_m)$ (Notice that in case $m = 1$ this vector contains 2 elements according to that explained in **step 19)**)
- $(\vec{b}_0, \dots, \vec{b}_{m-1})$ (Notice that in case $m = 1$ this vector contains 2 elements according to that explained in **step 19)**)
- $\vec{s} = (s_0, \dots, s_{m-1})$ (Notice that in case $m = 1$ this vector contains 2 elements according to that explained in **step 19)**)

Operation

1) If $m = 1$ set $m = 2$ (this change only applies to this argument).
2) Generate n random elements between 1 and q-1 and construct the vector $\vec{a}_0$.
3) Commit to the vector $\vec{a}_0$ using the **Commitment generation** primitive with the following inputs:
   - A random exponent $r_0 \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed: $\vec{a}_0$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_{A_0} = com_{ck}(\vec{a}_0; r_0)$.
4) Generate n random elements between 1 and q-1 and construct the vector $\vec{b}_m$.
5) Commit to the vector $\vec{b}_m$ using the **Commitment generation** primitive with the following inputs:
   - A random exponent $s_m \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed: $\vec{b}_m$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_{B_m} = com_{ck}(\vec{b}_m; s_m)$.
6) Define a new operation (we will denote it as $*$) that given two vectors, $(a_1, \dots, a_n)$ and $(d_1, \dots, d_n)$, does the following:

$$(a_1, \dots, a_n) * (d_1, \dots, d_n) = \sum_{j=1}^{n} a_j d_j y^j$$

   where y is the hash computed in **step 4)** of the Hadamard product argument.
7) Compute the values $d_k = \sum_{\substack{0 \le i,j \le m \\ j=(m-k)+i}} \vec{a}_i * \vec{b}_j$ with $k = 0, \dots, 2m$:

$$d_0 = \vec{a}_0 * \vec{b}_m$$
$$d_1 = \vec{a}_0 * \vec{b}_{m-1} + \vec{a}_1 * \vec{b}_m$$
$$d_2 = \vec{a}_0 * \vec{b}_{m-2} + \vec{a}_1 * \vec{b}_{m-1} + \vec{a}_2 * \vec{b}_m$$
$$\vdots$$

$$d_m = \sum_{i=0}^{m} \vec{a}_i * \vec{b}_i$$
$$d_{m+1} = \sum_{i=1}^{m} \vec{a}_i * \vec{b}_{i-1}$$
$$\vdots$$
$$d_{2m} = \vec{a}_m * \vec{b}_0$$

Define the vector $\vec{d} = (d_0, \dots, d_{2m})$.

8) Generate $2m + 1$ random elements between 1 and q-1 and construct the vector $\vec{t} = (t_0, \dots t_{2m})$. Set the element t_{m+1} of the vector to 0

9) Commit to each element of the vector $\vec{d}$ using the **Commitment generation** primitive with the following inputs:
   - The corresponding randomness t_i
   - List of elements to be committed: d_i (list with one element)
   - Commitment key $ck = (G_1, H)$

   The result is the commitment $c_{D_i} = com_{ck}(d_i; t_i)$.

10) After computing all the commitment define $\vec{c}_D$ as $\vec{c}_D = com_{ck}(\vec{d}; \vec{t}) = (c_{D_0}, \dots., c_{D_{2m}})$.

11) Concatenate the values of $\vec{c}_A$, $\vec{c}_B$,c_{A_0}, c_{B_m} and $\vec{c}_D$ in the following way:
   - For each element in $\vec{c}_A$ convert it to a string and concatenate all of them in a single value.
   - For each element in $\vec{c}_B$ convert it to a string and concatenate all of them in a single value.
   - Convert c_{A_0} to a string.
   - Convert c_{B_m} to a string.
   - For each element in $\vec{c}_D$ convert it to a string and concatenate all of them in a single value.

   Concatenate in a single value all the results obtained from all the steps above and compute a hash of the concatenation. Call the result $x = Hash(\vec{c}_A|\vec{c}_B|c_{A_0}|c_{B_m}|\vec{c}_D)$.

12) Given the set of vectors $(\vec{a}_0, \vec{a}_1, \dots, \vec{a}_m)$ and the hash x compute the vector $\vec{a}$ in the following way:

$$\vec{a} = \sum_{i=0}^{m} x^i \vec{a}_i$$

13) Given the set of values $(r_0, r_1, \dots, r_m)$ and the hash x compute the value r in the following way:

$$r = \sum_{i=0}^{m} x^i r_i$$

14) Given the set of vectors $(\vec{b}_0, \vec{b}_1, \dots, \vec{b}_m)$ and the hash x compute the vector $\vec{b}$ in the following way:

$$\vec{b} = \sum_{j=0}^{m} x^{m-j}\vec{b}_j$$

15) Given the set of values $(s_0, s_1, \dots, s_m)$ and the hash x compute the value s in the following way:

$$s = \sum_{j=0}^{m} x^{m-j}s_j$$

16) Given the set of values $(t_0, s_1, \dots, t_{2m})$ and the hash x compute the value t in the following way:

$$t = \sum_{k=0}^{2m} x^{k}t_k$$

Output

- Output $\vec{a}, r, \vec{b}, s, t$

11.1.10.5 Single value product argument

Input

- b
- $\vec{a} = (a_1, \dots, a_n)$
- $c_a = com_{ck}(\vec{a}; r)$
- $r \in \mathbb{Z}_q$

Operation

1) Given $\vec{a}$, compute the following values:

$$b_1 = a_1 \quad b_2 = a_1 a_2 \quad \cdots \quad b_n = \prod_{i=1}^{n} a_i$$

2) Generate n random exponents $d_1, \dots d_n \leftarrow \mathbb{Z}_q$ between 1 and q-1 and define the vector $\vec{d} = (d_1, \dots, d_n)$.

3) Commit to the vector $\vec{d}$ using the **Commitment generation** primitive with the following inputs:

   - A random exponent $r_d \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed: $\vec{b}_2$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   The result is the commitment $c_d = com_{ck}(\vec{d}; r_d)$

4) Define two values δ_1 and δ_n as $\delta_1 = d_1$, $\delta_n = 0$

5) Generate the random exponents $\delta_2, \dots, \delta_{n-1} \leftarrow \mathbb{Z}_q$ between 1 and q-1.

6) From $d_2, \dots d_n$ and $\delta_1, \delta_2, \dots, \delta_{n-1}$ compute the following values for $i = 1, \dots, n-1$:

$$\begin{gathered} -\delta_1 d_2 \\ -\delta_2 d_3 \\ \vdots \\ \delta_i d_{i+1} \\ \vdots \\ -\delta_{n-1} d_n \end{gathered}$$

7) Commit to the elements generated in the previous steps using **Commitment generation** primitive with the following inputs:
   - A random exponent $s_1 \in \mathbb{Z}_q$ between 1 and q-1.
   - List of elements to be committed:$(-\delta_1 d_2, \dots, -\delta_{n-1} d_n)$
   - Commitment key $ck = (G_1, \dots, G_{n-1}, H)$

   The result is the commitment $c_\delta = com_{ck}(-\delta_1 d_2, \dots, -\delta_{n-1} d_n; s_1)$
8) From $\delta_1, \delta_2, \dots, \delta_n$, $d_2, \dots d_n$ and $a_2, \dots a_n$ compute the following values for $i = 1, \dots, n-1$:

$$\begin{gathered} -\delta_2 - a_2\delta_1 - b_1 d_2 \\ -\delta_3 - a_3\delta_2 - b_2 d_3 \\ \vdots \\ -\delta_{i+1} - a_{i+1}\delta_i - b_i d_{i+1} \\ \vdots \\ -\delta_n - a_n\delta_{n-1} - b_{n-1} d_n \end{gathered}$$

9) Commit to the elements generated in the previous steps using the **Commitment generation** primitive with the following inputs:
   - A random exponent $s_x \in \mathbb{Z}_q$ between 1 and q-1
   - List of elements to be committed:$(-\delta_2 - a_2\delta_1 - b_1 d_2, \dots, -\delta_n - a_n\delta_{n-1} - b_{n-1} d_n)$
   - Commitment key $ck = (G_1, \dots, G_{n-1}, H)$

   The result is the commitment $c_\Delta = com_{ck}(\delta_2 - a_2\delta_1 - b_1 d_2, \dots, \delta_n - a_n\delta_{n-1} - b_{n-1} d_n; s_x)$
10) Convert the values of c_a, b ,c_d, c_δ and c_Δ to a string and concatenate all of them in a single value. Compute a hash of the concatenation and call the result $x = Hash(c_a|b|c_d|c_\delta|c_\Delta)$.
11) Given $\vec{a}$, $\vec{d}$, r, r_d and x, compute the following values:

$$\begin{gathered} \tilde{a}_1 = x a_1 + d_1 \\ \vdots \\ \tilde{a}_n = x a_n + d_n \\ \tilde{r} = x r + r_d \end{gathered}$$

12) Given $\vec{b}$, $\delta_1, \dots, \delta_n$, s_1, s_x and x, compute the following values:

$$\begin{gathered} \tilde{b}_1 = x b_1 + \delta_1 \\ \vdots \\ \tilde{b}_n = x b_n + \delta_n \\ \tilde{s} = x s_x + s_1 \end{gathered}$$

Output

- Output $(\tilde{a}_1, \dots, \tilde{a}_n)$, $(\tilde{b}_1, \dots, \tilde{b}_n)$, $\tilde{r}, \tilde{s}$.

11.1.11 Mixing proof verifier

The description of the verification of the mixing zero-knowledge proof is described in the original paper from Stephanie Bayer and Jens Groth [3].

Input

- Input ciphertexts: $\vec{C}$
- Output ciphertexts: $\vec{C}'$

- Encryption parameters
- Commitment parameters
- Public key
- Mixing proof:
    - initialMessage → $\vec{c}_A$
    - firstAnswer → $\vec{c}_B$
    - secondAnswer:
        - msgPA → represents the initial message of Product Argument
            - commitmentPublicB → c_b
            - iniHPA → Initial message of Hadamard Product Argument
                - commitmentPublicB → $\vec{c}_B$
            - ansHPA → Answer of Hadamard Product Argument
                - initial → Initial message of Zero Argument
                    - commitmentPublicA0 → c_{A_0}
                    - commitmentPublicBM → c_{B_m}
                    - commitmentPublicD → $\vec{c}_D$
                - answer → Answer of Zero Argument
                    - exponentsA → $\vec{a}$
                    - exponentsB → $\vec{b}$
                    - exponentR → r
                    - exponentS → s
                    - exponentT → t
                - iniSVA → represents the initial message of Single Value Product Argument
                    - commitmentPublicD → c_d
                    - commitmentPublicLowDelta → c_δ
                    - commitmentPublicHighDelta → c_Δ
                - ansSVA → represents the answer of Single Value Product Argument.
                    - exponentsTildeA → $\tilde{a}_1, \ldots, \tilde{a}_n$
                    - exponentsTildeB → $\tilde{b}_1, \ldots, \tilde{b}_n$
                    - exponentsTildeR → $\tilde{r}$
                    - exponentsTildeS → $\tilde{s}$
            - iniMEBasic → initial message of multi-exponentiation argument
                - commitmentPublicA0 → c_{A_0}
                - commitmentPublicB → $\{c_{B_k}\}_{k=0}^{2m-1}$
                - ciphertextsE → $\{E_k\}_{k=0}^{2m-1}$

- ansMEBasic → answer of multi-exponentiation argument
  - exponentsA → $\vec{a}$
  - exponentR → r
  - exponentB → b
  - exponentS → s
  - randomnessTau → τ

Operation

The paper presents all the zero knowledge arguments as an interactive protocol between the prover and the verifier. Our implementation uses a non-interactive approach using the Fiat-Shamir transformation to modify the challenges so that they can be obtained using a hash function. The parameters used as input for the hash functions are the statement in the order presented in the paper and the values sent in the initial message are also preserving the order from the paper. We do not separate the values, just append the new values. More precisely:

- Shuffle argument:
  - challenge x → $x^{SA} = H(\vec{C}|\vec{C}'|initialMessage)$
  - challenge y → $y^{SA} = H(\vec{C}|\vec{C}'|initialMessage|firstAnswer)$
  - challenge z → $z^{SA} = H(\vec{C}|\vec{C}'|initialMessage|firstAnswer|1)$
- Product argument does not have any challenge
- Hadamard product argument:
  - challenge x → $x^{HPA} = H(\vec{c}_A|c_b|iniHPA)$
  - challenge y → $y^{HPA} = H(\vec{c}_A|c_b|iniHPA|1)$
- Zero argument:
  - challenge x → $x^{ZA} = H(\vec{c}_A|\vec{c}_B|ansHPA.initial)$
- Single value product argument:
  - challenge x → $x^{SVPA} = H(c_a|b|iniSVA)$
- Multi-exponentiation argument:
  - challenge x → $x^{MA} = H(\vec{C}'|C|\vec{c}_A|iniMEBasic)$

It is also important to note that the operator | means concatenation. To append the values from a multi-variable element from the `proof.json` file, the process is to concatenate the elements in the same order presented in the paper, which is also the order in which we have written it at the beginning of the section. For instance, the element ansHPA.initial would be represented as:

$$ansHPA.initial = c_{A_0}|c_{B_m}|\vec{c}_D$$

1) Validate the Hadamard product argument
   - Check that $c_{B_2}, \dots, c_{B_{m-1}} \in \mathbb{G}$ where $\vec{c}_B$ is taken from msgPA.iniHPA.commitmentPublic
   - Define $\vec{c}_D = \vec{c}_A^{\,y} \cdot \vec{c}_B$, where $\vec{c}_A$ is taken from the initialMessage, $\vec{c}_B$ from firstAnswer, and y^{SA} is the challenge computed in the Shuffle argument
   - Define $\vec{c}_{-z} = \left(com_{ck}(-z, \dots, -z; 0), \dots, com_{ck}(-z, \dots, -z; 0)\right)$.
   - Define $\vec{c}_A^{\,PArg} = \vec{c}_D \vec{c}_{-z}$.
   - Check that $c_{B_1} = c_{A_1}^{PArg}$
   - Check that $c_{B_m} = c_b$ where c_b is taken from msgPA.commitmentPublicB.
   - Accept if the zero argument is valid.
2) Validate Zero argument
   - Compute the commitment to $\vec{a}$ (ansHPA.answer.exponentsA) using the randomness r (ansHPA.answer.exponentR) and the commitment key: $com_{ck}(\vec{a}; r)$
   - Compute the commitment to $\vec{b}$ (ansHPA.answer.exponentsB) using the randomness s (ansHPA.answer.exponentS) and the commitment key: $com_{ck}(\vec{b}; s)$
   - Compute the commitment to $\vec{a} * \vec{b}$ (ansHPA.answer.exponentsB) using the randomness t (ansHPA.answer.exponentT) and the commitment key: $com_{ck}(\vec{a} * \vec{b}; t)$.
   - Define $c_{D_i} = c_{B_i}^{x^i}$ and $c_D = \prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i}$, where $\vec{c}_B$ is taken from msgPA.iniHPA.commitmentPublicB and x is the challenge computed in the Hadamard Product argument.
   - Define $\vec{c}_B$ of the Zero argument as $\vec{c}_B = (c_D, c_{D_1}, \dots, c_{D_{m-1}})$. Notice that if $m = 1$ the vector will have only two elements.
   - Compute the challenge x^{ZA} as $Hash(\vec{c}_A|\vec{c}_B|ansHPA.initial) = Hash(\vec{c}_A|\vec{c}_B|c_{A_0}|c_{B_m}|\vec{c}_D)$.
   - Define $c_{-1} = com_{ck}(-\vec{1}; 0)$ where $-\vec{1}$ is the vector of n elements filled with the value -1.
   - Compute $\vec{c}_A^{\,PArg}$ in the same way as it was computed during the validation of the Hadamard Product Argument.
   - Define $\vec{c}_A$ of the Zero argument as $\vec{c}_A = (c_{-1}, c_{A_2}^{PArg}, \dots, c_{A_m}^{PArg})$. Notice that if $m = 1$ the vector will have only two elements.
   - Compute $\prod_{i=0}^{m} c_{A_i}^{x^i}$
   - Compute $\prod_{j=0}^{m} c_{B_j}^{x^{m-j}}$
   - Compute $\prod_{k=0}^{2m} c_{D_k}^{x^k}$ where $\vec{c}_D$ is taken from ansHPA.initial.commitmentPublicD.
   - Check that the following equations hold:

$$\prod_{i=0}^{m} c_{A_i}^{x^i} = com_{ck}(\vec{a}; r) \quad \prod_{j=0}^{m} c_{B_j}^{x^{m-j}} = com_{ck}(\vec{b}; s) \quad \prod_{k=0}^{2m} c_{D_k}^{x^k} = com_{ck}(\vec{a} * \vec{b}; t)$$

- Check that:
  - $c_{A_0}, c_{B_m} \in \mathbb{G}$
  - $\vec{c}_D \in \mathbb{G}^{2m+1}$
  - $c_{D_{m+1}} = com_{ck}(0; 0)$
  - $\vec{a}, \vec{b} \in \mathbb{Z}_q^n$
  - $r, s, t \in \mathbb{Z}_q$

3) Validate the Single value product argument
   - $c_d, c_\delta, c_\Delta \in \mathbb{G}$
   - $\tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_n, \tilde{b}_n, \tilde{r}, \tilde{s} \in \mathbb{Z}_q$
   - Compute $com_{ck}(\tilde{a}_1, \ldots, \tilde{a}_n; \tilde{r})$ using the values stored in ansHPA.ansSVA.exponentsTildeA and in ansHPA.ansSVA.exponentsTildeR.
   - Compute the challenge $x^{SA} = Hash(\vec{C}|\vec{C}'|\vec{c}_A)$ where $\vec{c}_A$ is taken from the initialMessage.
   - Compute the challenge $y^{SA} = Hash(\vec{C}|\vec{C}'|\vec{c}_A|\vec{c}_B)$ where $\vec{c}_B$ is taken from the firstAnswer.
   - Compute the challenge $z^{SA} = Hash(\vec{C}|\vec{C}'|\vec{c}_A|\vec{c}_B|1)$.
   - Define the array of values $\vec{a} = (a_1, \ldots, a_N)$ where N is the number of ciphertexts that have been mixed. Compute the array $\vec{b} = (b_1, \ldots, b_N) = (x^{a_1}, \ldots, x^{a_N})$ using x^{SA}.
   - Compute the value $b = \prod_{i=1}^{N}(y \cdot a_i + b_i - z)$ using y^{SA} and z^{SA}.
   - Compute the challenge $x^{SVPA} = Hash(c_a|b|c_d|c_\delta|c_\Delta)$. c_d is taken from ansHPA.iniSVA.commitmentPublicD, c_δ from ansHPA.iniSVA.commitmentPublicLowDelta and c_Δ from commitmentPublicHighDelta.
   - Compute $com_{ck}\ (x\tilde{b}_2 - \tilde{b}_1\tilde{a}_2, \ldots, x\tilde{b}_n - \tilde{b}_{n-1}\tilde{a}_n; \tilde{s})$ using the value stored in ansHPA.ansSVA.exponentsTildeA, ansHPA.ansSVA.exponentsTildeB, ansHPA.ansSVA.exponentsTildeS and the challenge x^{SVPA}.
   - Check that the following equations hold:

$$c_a^x c_d = com_{ck}(\tilde{a}_1, \ldots, \tilde{a}_n; \tilde{r}) \quad c_\Delta^x c_\delta = com_{ck}\ (x\tilde{b}_2 - \tilde{b}_1\tilde{a}_2, \ldots, x\tilde{b}_n - \tilde{b}_{n-1}\tilde{a}_n; \tilde{s})$$
$$\tilde{b}_1 = \tilde{a}_1 \quad \tilde{b}_n = xb$$

4) Validate the Product argument
   - Check if $c_b \in \mathbb{G}$
   - The Product argument is valid if both the Hadamard product argument and the Single value product argument are convincing.

5) Validate Multi-exponentiation argument
   - Check that $c_{A_0}, c_{B_0}, \ldots, c_{B_{2m-1}} \in \mathbb{G}$
   - Check that $E_0, \ldots, E_{2m-1} \in \mathbb{H}$
   - Check that $\vec{a} \in \mathbb{Z}_q^n$ and $r, b, s, \tau \in \mathbb{Z}_q$
   - Obtain c_{B_m} from iniMEBasic.commitmentPublicB and check that $c_{B_m} = com_{ck}(0; 0)$.

- Compute the challenge $x = Hash(\vec{C}|\vec{C}'|\vec{c}_A)$ where $\vec{c}_A$ is taken from the initialMessage.
- Compute $C = \prod_{i=1}^{N} C_i^{x^i}$.
- Obtain E_m from iniMEBasic.ciphertextsE and check that $E_m = C$.
- Obtain c_{A_0} from iniMEBasic.commitmentPublicA0
- Obtain $\vec{c}_A$ from the firstAnswer and compute $x^{MA} = H\left(\vec{C}|\vec{C}'|\vec{c}_A|iniMEBasic\right) = H\left(\vec{C}|\vec{C}'|\vec{c}_A|c_{A_0}|\{c_{B_k}\}_{k=0}^{2m-1}|\{E_k\}_{k=0}^{2m-1}\right)$.
- Define $\vec{x} = (x, x^2, \dots, x^m)$ and compute $c_{A_0}\vec{c}_A^{\vec{x}}$ using x^{MA}.
- Check that $com_{ck}(\vec{a}; r)$ where $\vec{a}$ is taken from ansMEBasic.exponentsA and r from ansMEBasic.exponentR.
- Obtain $\{c_{B_k}\}_{k=0}^{2m-1}$ from iniMEBasic.commitmentPublicB, b from ansMEBasic.exponentB and s from ansMEBasic.exponentS and check that:

$$c_{B_0} \prod_{k=1}^{2m-1} c_{B_K}^{x^k} = com_{ck}(b; s)$$

- Obtain $\{E_k\}_{k=0}^{2m-1}$ from iniMEBasic.ciphertextsE and $\vec{a}$ from ansMEBasic.exponentsA.
- Compute the ElGamal encryption of G^b using the public key received as input and the randomness τ, stored in ansMEBasic.randomnessTau: $\mathcal{E}_{pk}(G^b; \tau)$.
- Check that:

$$E_0 \prod_{k=1}^{2m-1} E_k^{x^k} = \mathcal{E}_{pk}(G^b; \tau) \prod_{i=1}^{m} \vec{C}_i^{x^{m-i}\vec{a}}$$

Bayer and Groth modification for m=1

Since the option of appending dummy votes and later remove it was discarded from the beginning, the shuffle argument as it was defined by Bayer and Groth required to be modified to accommodate the proof for cases in which the amount of received ballots was a prime number. The solution was to perform a small modification allowing the protocol to run with m=1.

This change only affected the product proof argument. More precisely, the Hadamard product argument required by the product proof argument. This Hadamard product argument receives as input:

- Arrays of exponents: $\vec{a}_1, \dots, \vec{a}_m$
- Array of exponents: $\vec{r}$
- Array of exponents: $\vec{b}$
- An exponent s

such that:

$$c_{a_1} = com(\vec{a}_1; r_1) \dots c_{a_m} = com(\vec{a}_m; r_m)$$

$$c_b = com(\vec{b}; s)$$

$$\vec{b} = \prod_{i=0}^{m} \vec{a}_i$$

The first step of the protocol for proving this argument was to compute the initial message. In the paper, it was required to compute $\vec{b}_{m-1} = \vec{a}_1 \cdot \vec{a}_{m-1}$. However, when m=1 this computation requires the use of $\vec{a}_0$ but there is no such value, so the protocol does not work.

To resolve this issue, when m=1, the prover generates a commitment $c_{a_2} = com(\vec{a}_2; r_2)$ being $\vec{a}_2$ an array with *n* positions filled with 1s and $r_1 = 0$.

This way, the proof is consistent, and the generated commitment is easily checked. Moreover, every verifier can implement it in the same way as it does not contain any new randomly generated value.

11.1.12 Group Element generation

Input

- Mathematical group (p, q, g)

Operation

- Generate a random exponent $r \in \mathbb{Z}_q$ between 1 and q-1.
- Exponentiate the generator g to the random exponent: $H = g^r \ mod \ p$

Output

- The group element H

11.1.13 Commitment generation

Input

- Random exponent r
- List of elements to be committed: $\vec{a} = (a_1, \dots, a_n) \in \mathbb{Z}_q^n$
- Commitment key $ck = (G_1, \dots, G_n, H)$

Operation

- Compute the exponentiation of H to r: H^r
- For each a_i where $i = 1, \dots, n$ compute the exponentiation $G_i^{a_i}$.
- Multiply all the exponentiations and obtain the commitment:

$$com_{ck}(\vec{a}; r) = com_{ck}(a_1, \dots, a_n; r) = H^r \prod_{i=1}^{n} G_i^{a_i}$$

Output

- The commitment: $com_{ck}(\vec{a}; r)$

11.1.14 ElGamal encryption

Input

- The mathematical group defined by (p, q, g)
- Public key $(pk_1, pk_2, \dots, pk_k)$ (array of length $k > 0$)
- Plaintext $(m_1, m_2, \dots, m_n)$ of at most k element and at least 1 element.

Operation

- The mathematical group defined by (p, q, g)
- Generate a random exponent r between 1 and $q - 1$.
- Generate the first element of the ciphertext $C_0 = g^r \ mod \ p$.
- If the size of the plaintext is smaller than the size of the public key, compute
  - $pk_n = pk_n + \ pk_{n+1} + pk_{n+2} + \cdots + \ pk_k$ *where the value of pk_n on the right is the old one and the value on the left is the one used from this point onwards.*
- For each element in the plaintext, compute the following elements of the cihpertext: $C_i = (pk_i)^r \cdot m_i \ mod \ p$ for $i = 1, \dots n$.

Output

- The generated random exponent r
- The ciphertext $(C_0, C_1, \dots, C_n)$

11.2 LDAP API

```
final LdapHelper ldapHelper = new LdapHelper();

String subjectDn =_certificate.getSubjectX500Principal().getName();
try {
      String subjectCn =
          ldapHelper.getAttributeFromDistinguishedName(subjectDn,
                X509CertificateConstants.COMMON_NAME_ATTRIBUTE_NAME);
      String subjectOrgUnit =
          ldapHelper.getAttributeFromDistinguishedName(subjectDn,
                X509CertificateConstants.ORGANIZATIONAL_UNIT_ATTRIBUTE_NAME);
      String subjectOrg =
          ldapHelper.getAttributeFromDistinguishedName(subjectDn,
                X509CertificateConstants.ORGANIZATION_ATTRIBUTE_NAME);
       String subjectCountry =
          ldapHelper.getAttributeFromDistinguishedName(subjectDn,
                X509CertificateConstants.COUNTRY_ATTRIBUTE_NAME);
```

11.3 Coding and conversions

During the proof generation, the values are obtained and converted from one class to another. The following lines describe how these conversions are done:

- *String to byte[]:* Strings are transformed into byte arrays by the method getBytes() of the String class using `StandardCharsets.UTF_8`.
- *BigInteger to byte[]:* BigIntegers are transformed by using toString() method first, and then use getBytes().
- *byte[] to BigInteger:* byte arrays are transformed into BigIntegers using the BigInteger constructor: BigInteger(byte[] input).
- *byte[] to Base64 string:* byte arrays are encoded base64 strings using the method defined either in java.util.Base64 (`Base64.getEncoder().encodeToString`) or in org.apache.commons.codec.binary.Base64 (`Base64.encodeBase64String`).
- *Base64 string to byte[]:* strings encoded in base64 are decoded using the method defined either in java.util.Base64 (`Base64.getDecoder().decode`), in org.apache.commons.codec.binary.Base64 (`Base64.decodeBase64`) or in org.bouncycastle.util.encoders.Base64 (`Base64.decode`)
- *byte[] to Base64 byte[]:* byte arrays are encoded base64 byte array[] using the method defined either in org.apache.commons.codec.binary.Base64 (`Base64.encodeBase64`) or in java.util.Base64 (`Base64.getEncoder().encode`).

11.4 Data concatenation

```
public byte[] concatenate(String... data) {
      return StringUtils.join(data).getBytes(StandardCharsets.UTF_8);
}
```

11.5 Cryptographic algorithms

These are the cryptographic algorithms used by the online voting system:

- **Asymmetric encryption**: ElGamal, key length = 2048 bits
- **Symmetric encryption:** AES128-GCM
- **Digital signature:** RSA-PSS, SHA2-256, key length = 2048 bits
- **Hash:** SHA2-256
- **Message Authentication Code:** HMAC with SHA2-256

11.6 EV Solution Intellectual Property Rights Notice (the Notice)

Scytl sVote is part of a larger system called EV Solution, developed under the "Framework Agreement" entered into by and between Post CH Ltd (Swiss Post) and Scytl Secure Electronic Voting, S.A. (Scytl) on September 30th 2015.

Parts of this EV Solution system and other relevant details are defined below.

11.6.1 Definitions

The following terms shall have the meanings specified below:

"**EV Solution**" means an online voting system consisting of the Scytl Standard Software (also referred to as Scytl sVote or Scytl Online Voting 2.0) in combination with the Swiss Post-Scytl Software, and all the associated middleware provided by Scytl as a bundle with the Scytl Standard Software and the Swiss Post-Scytl Software. Software below middleware (e.g. Linux OS and Windows OS and Oracle software) that are needed to run the EV Solution are not part of the EV Solution.

"**Intellectual Property Rights**" or "**IPRs**", for the purposes of this Notice and pursuant to the Framework Agreement, means copyright and patent rights (if any), know-how and trade secrets, performance rights and entitlements to such rights.

"**Scytl Online Voting 2.0**" is the brand name that was used to identify Scytl Standard Software in the market.

"**Scytl Standard Software**" means all software developed by Scytl for the EV Solution, whose architecture, specifications and capabilities are described in Scytl sVote documents, excluding Swiss Post-Scytl Software and software developed by Scytl independently to the EV Solution.

"**Software**" means software code (source code and object code), user interfaces and documentation (preparatory documentation and manuals) and including releases and patches etc.

"**Scytl sVote**" means the registered trademark proprietary to Scytl, that identifies Scytl Standard Software in the market.

"**Swiss Post-Scytl Software**" means the software developed for the EV Solution (excluding Scytl Standard Software) pursuant to the Framework Agreement. Swiss Post-Scytl Software comprises of the following:

i. Key Translation Module: A mapping service that translates external IDs to internal IDs for specific entities so that external systems can integrate with sVote.

ii. Swiss Post Integration Tools: A group of applications that allow the integration between Swiss Post's applications and sVote through file conversions.

iii. Swiss Post Voting Portal Frontend: Frontend application that guides the voters throughout all the voting steps enabling them to successfully cast a vote for a particular election.

11.6.2 Copyright notice

11.6.2.1 Scytl Standard Software

All intellectual property rights in the Scytl Standard Software are Scytl's sole property. Scytl owns and shall retain all rights, title and interest in and to the Scytl Standard Software. Scytl Standard Software is licensed to Swiss Post under the terms and conditions described in the Framework Agreement.

11.6.2.2 Swiss Post-Scytl Software

All intellectual property rights in the Swiss Post-Scytl Software are the joint property of Scytl and Swiss Post (Joint IP).

11.6.2.3 EV Solution

All intellectual property rights in the EV Solution other than Joint IP will be owned by Scytl or by third parties as applicable.