Scytl sVote

Protocol Specifications

Software version 2.1

Document version 5.1

Scytl – Secure Electronic Voting

STRICTLY CONFIDENTIAL

Table of contents

List of figures

List of tables

1 Introduction

This document gives a detailed description of the cryptographic protocol implemented in Scytl sVote. The protocol provides end-to-end encryption, vote secrecy and both **individual** and **universal verifiability** to the product.

- **Individual verifiability** is accomplished by sending to the voter a Choice Return Code for each possible voting option the voter has selected during the voting process. These Choice Return Codes are calculated and returned to the voter by the voting server using the encrypted vote sent by the voter and without decrypting the vote. If the voter agrees with the Choice Return Codes[1] (with the assistance of a voting card), sends a ballot confirmation key (Ballot Casting Key). The server verifies the Ballot Casting Key an if it is correct, calculates the corresponding Vote Cast Return Code that confirm to the voter that the confirmed vote is included in the Ballot Box.
- **Universal verifiability** is achieved using a set of independent components named Control Components. The use of these independent components to achieve universal verifiability is a specific proposal made by the Swiss Federal Chancellery in the Electronic Voting Ordinance (VEleS) [1] [2].

This approach is based on a trust model in which the privacy and auditability of the election is not based on the trustworthiness of the voting client or voting servers components, but on the trustworthiness of at least one control component within a group of control components.

Furthermore, the individual verifiable proof should also allow voters to ensure that the universal verifiability mechanism considers their votes (e.g., the Choice Return Codes should be generated by Control Components).

The document describes a voting system in which voters can choose among pre-defined options. In case the voter cast a vote with a write-in, the individual verification process only confirms the presence of a write-in in the vote, but it does not prove the write-in value. The rest of the protocol properties are preserved in this case.

1.1 Document organization

This document is organized as follows:

- **Section 1: Introduction.** This section is a short introduction to the protocol and the trust model.
- **Section 2: Solution overview.** This section is a summary of the general solution with all the involved components.

[1] In some parts of the sVote code, this concept might be referred to as “Return Code” instead of “Choice Return Code

- **Section 3: System configuration process**. This section also defines the system certificate hierarchy and outlines the steps required for certificates generation. These are the certificates used by the voting system that are shared among different election events.
- **Section 4: Election configuration process**. This section defines the data, keys and credentials that are specific for each election and how they are generated. It includes the generation of the keys relevant for the individual verification functionality of the protocol, and the keys relevant for the complete verification functionality. At the beginning of this section, there are some tables that summarize the notation regarding keys and variables introduced throughout the section and used along the document.
- **Section 5: Voting phase**. This section explains the different steps of the voting phase: (authentication, send a vote and confirm a vote) and the algorithms involved in them.
- **Section 6: Counting phase**. This section describes all the activities performed during the counting process (Cleansing, Mixing and Decryption) with special focus on the Ballot Box export process.
- **Section 7: Audit phase**. This section specifies how to audit the election process.
- **Section 8**: **References.** This section specifies the literature used as a reference throughout the document.
- **Section 9: Appendix**. This section details the pre-computations that can be done both in the voting client and during configuration to reduce the time needed to cast a vote. It also describes all the cryptographic primitives used by the voting protocol. Throughout the document these primitives are referenced when needed.

2 Solution overview

The following diagram is an overview of the components involved in the voting system and the communication channels established between them:

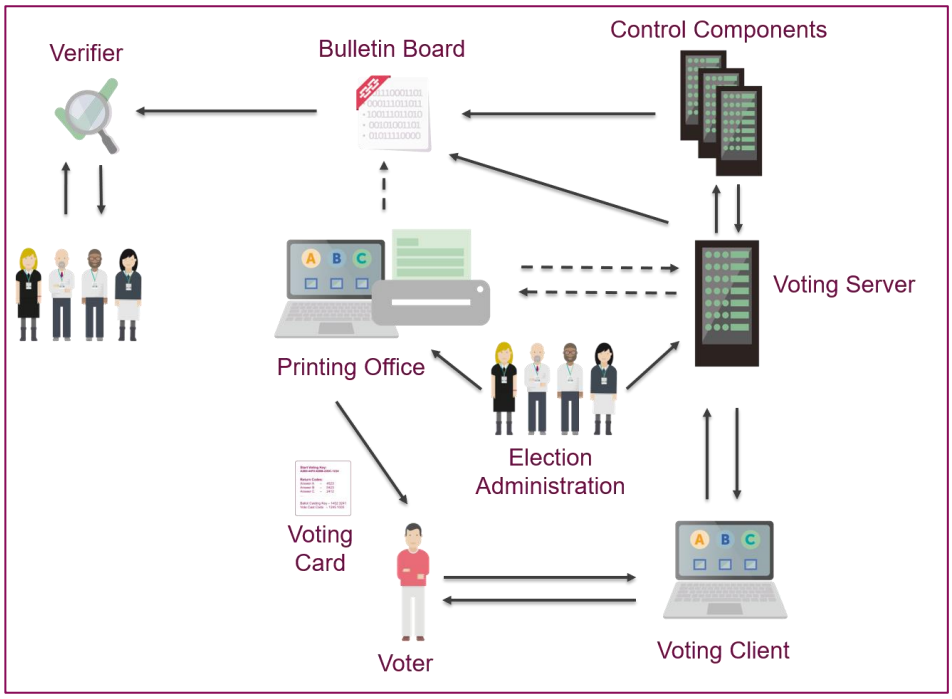

Figure 1 - Overview of the voting system components and communication channels

Each of these components is defined as follows:

- **Voter**: Is the user of the system. Prior to the election, the voter receives a Voting Card with the keys and codes to vote and verify his/her vote.
- **Voting Client**: This is the device used by the voter to cast his/her vote. This component is also named as Client Context in the protocol description.
- **Voting Server**: It authenticates the voter and receives, processes and stores in the Ballot Box the votes cast by the voter. 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 and 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.

The Voting Server also hosts the following election configuration and management modules for the Election administrators not accessible to voters:

- Administration Portal performs non-cryptographic operations (configure the ballot, define the Electoral Board members, etc.) and is used to introduce the election configuration information such as the election name, election calendar, candidates, electoral roll, etc.
- Election Configuration Services interact with the Administration Portal, the Print Office domain and the Control Components domain during the configuration phase. They provide election information to all the components of the different domains and facilitate the communication between the Print Office and the Control Components domains for generating the election cryptographic-related information. The Election Configuration Services use the Secure Data Manager (SDM online) software component to implement the above-mentioned functionalities.

- **Control Components (CC)**: According to the VEleS annex [2], they can be implemented as:
  - A group of people: People are considered only for protecting voter privacy during the vote decryption process. In this case, it is possible to set up a group of at least 4 people for keeping in smartcards the shares of the secret key.
  - Standard computers: At least 4 computers with different Operating Systems to ensure that they do not share a common threat.
  - Hardware Security Modules: At least 2 HSM[2] from different vendors with Common Criteria Evaluation Assurance level 4 (CC EAL4) or the level 3 certification of the Federal Information Processing Standard (**FIPS**) Publication **140-2**. Control Components can be combined in one or more groups. All the Control Components in a group should collaborate to perform their assigned voting protocol function and any attempt to abuse the system should always be detected if at least one component of the group is honest.

[2] A hardware security module (HSM) is a physical computing device that safeguards and manages digital keys for strong authentication and provides crypto processing.

The approach presented in this document considers standard computers and distinguishes between two types of Control Components:

- Choice Return Codes Control Components (CCR): They implement the generation of the Choice Return Codes and the Vote Cast Return Code but using a distributed approach. These components work in parallel and the results of their cryptographic operations are combined to obtain the final Choice Return Codes and Vote Cast Return Code.
- Mixing Control Components (CCM): These Control Components implement the shuffling and decryption of the votes during the counting process and are also 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 is based on implementing these mix-nodes using 4 Control Components. With the aim of distributing the decryption process across them, each mix-node performs a partial decryption in addition to shuffle and re-encryption. Additionally, the last mix-node (CCM_4) decrypts the votes after mixing using an Electoral Board key reconstructed using a secret sharing scheme. Only the first three nodes (CCM_1, CCM_2, CCM_3) partially decrypt the votes using its own partial decryption key that does not need to be reconstructed.

The last Mixing node (CCM_4) also needs to interact with the Key Reconstruction module to reconstruct the Election Administrators keys that allow partial decryption and digitally signing of the information managed by this Control Component:

- Key Reconstruction reconstructs the election keys required for protecting the integrity of the election information (Administration Board key) and the privacy of the votes (Electoral Authority key). The reconstruction is done from the shares provided by the members of the Board that custodies the keys. This module uses the software component called offline Secure Data Manager for performing the secret sharing scheme that reconstructs the key.

- **Print Office**: It is responsible for generating, printing and delivering the voting cards to the voters as well as for generating the required election keys. The generation is done in physically isolated infrastructure by the following modules.
  - Voting Card Generation is used to generate the voter's credentials (Voting Cards) interacting indirectly (through the Voting Server) with the Control Components.
  - Election Keys Generation generates the election keys required for protecting the integrity of the election configuration information (Administration Board key), the privacy of the votes (Electoral Board Authority key) and the integrity of the information generated and used during the voting phase (Authentication Token, Receipt, etc). The

Administration Board key and the Electoral Board Authority key are generated using a secret sharing scheme.

Both modules use the software component called offline Secure Data Manager (SDM).

- **Election Administrators**: They are responsible for generating the election configuration, verifying it, computing the results and publishing them. We distinguish between:
    - Administration Portal role that performs non-cryptographic operations (configure the ballot, define the Electoral Board members, etc.)
    - Administration Board Authority that uses the offline SDM from the Election Keys Generation module in the Print Office component, generates all the cryptographic information to ensure the integrity and security of the voting process. The Administration Board is also used from the Key Reconstruction by the last Control Component. This authority owns a digital signature key pair whose private key is shared among the Board members and is used to sign both the configuration and the results of the last Control Component execution.
    - Electoral Board Authority using the Election Keys Generation module in the Print Office component, generates all the cryptographic information to privacy of the voting process (i.e., the election keys). This entity owns a key pair whose private key is shared among the Board members and is used to partially decrypt the votes in the last Control Component execution.
- **Global Bulletin Board (Audit System)**: It is the global repository used to store all the audit information from the different modules that will be required to verify the election process. It stores election configuration, votes, confirmations and keeps track of all the actions performed by each entity. The Bulletin Board 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 (named offline Secure Data Manager) created after the execution of the Voting Card Generation and Election Keys Generation modules in the Print Office component, where all the election configuration is stored signed by the Administration

Board. The initial folder structure and contents are provided by the Administration Portal and Election Configuration Services modules of the Voting Server. The Print Office completes the contents of this folder with the generated election keys, and digitally signs it.

- **Verifier**: It is the component used to verify the correctness of the entire election process, the integrity of the data processed through different voting system components, and that these processes are accurate and fair. Using the information stored in the global Bulletin Board, the Verifier:
    - Ensures that the configuration sealed (digitally signed) by the Print Office using the Voting Card Generation and the Election Keys Generation, is the configuration used during the voting phase and that has not been altered after it has been signed.
    - Ensures that the behaviour of each component is the expected one.
    - Ensures that all the encrypted votes present in the Ballot Box correspond to the encrypted votes cast by legitimate voters during the voting phase.
    - Ensures that all the encrypted votes present in the Ballot Box correspond to votes that have been validated by the voters and processed by the Control Components.
    - Ensures that no votes processed by the Control Components have been deleted from Ballot Box.
    - Ensures that all the content of the valid encrypted votes that are present in the Ballot Box at the end of the voting phase are part of the tally.

  It is also important that all these verification processes are carried out without compromising the privacy of any voter. The details of the verifier are available in [3].

The Secure Data Manager (SDM) has been mentioned several times in the modules description, but it must not be identified as an architectural module of the voting system protocol. The SDM is a software component that provides the required cryptographic protocol operations to the modules. Therefore, it is not a standalone module that receives calls from other modules, but a software component implementing the cryptographic parts of the protocol used for election configuration set-up and secret-sharing key management.

When the SDM is called inside a module deployed in an isolated environment (e.g., Voting Card Generation), it is taken that it uses the SDM offline functionalities. Otherwise, it is considered that the module calls the SDM online functionalities that facilitates the communication with modules that are calling the SDM offline.

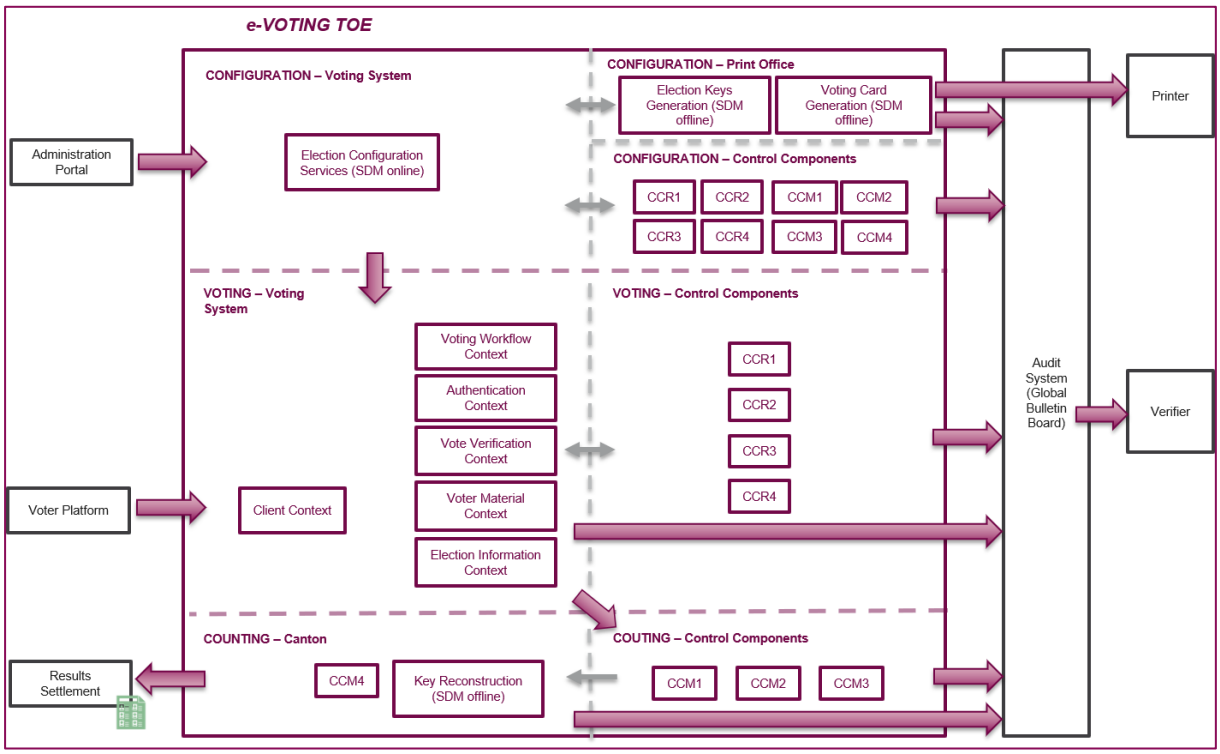

Figure 2 - Detailed view of the voting system modules and their interactions

As shown in Figure 2, the modules previously defined, interact during the voting protocol and assume the following phases:

- **Configuration**: This phase takes care of those activities related to the system and election configuration and provides all the data that will be required by other modules during the election, including codes, certificates, keys and passwords. The codes the voter will use when voting (Start Voting Key, Ballot Casting Key, Vote Cast Return Code and Choice Return Codes for each voting option) need to be sent to each voter before the election starts. The components involved in this phase are the Print Office, the Voting Server, the Control Components and the Election Administration.
- **Voting**: During the voting phase, voters access the voting application using their credentials, then they are presented with their ballot, select their option (s) and cast their vote. This phase needs the interaction between the voter, the Voting Client, the Voting Server and the Control Components.
- **Counting**: It comprises the modules for Cleansing, Mixing and Decryption of votes. The tallying is not considered as part of the solution and it is expected to be performed by the authorities.
    - **Cleansing**: Its main task is to validate votes in the Ballot Box before Mixing. Confirmed votes are kept in a cleansed Ballot Box without any reference to the voter (no Voting Card ID (vcd_{id}) and no signature) so they cannot be traced back when decrypted. This

process is done by the Voting Server and can be repeated by the auditors during the verification phase.

- **Mixing** and **Decryption**: These processes are executed sequentially in each Control Component. The mixing process performs a shuffle and re-encryption on the votes to break the link between the votes as they were stored in the Ballot Box (which can be traced to the Voting Card IDs) and the votes to be decrypted. The decryption computed by the Control Components is indeed a partial decryption since the votes are not fully decrypted unless all the Control Components participate in the process. The three first Control Components CCM_1, CCM_2 and CCM_3 partially decrypt their votes using their own CCM_j Mixing private key (x_j). However, the last partial decryption performed in CCM_4 uses the reconstructed Electoral Board private key (EB_{sk}). In this last Control Component $CCM_4$the Administration Board key is also reconstructed to sign its outputs.

2.1 Mapping solution to VEleS Swiss regulation

Protocol	VEleS Swiss regulation	Trust assumption
Voters	Voters	Significant proportion of voters are non-trustworthy
Voting Client	User platform	Untrustworthy for individual and complete verifiability, trustworthy for privacy
Voting Card	Trusted technical aids for voters	Trustworthy
Voting Server	System (server-side)	Untrustworthy
Print Office (offline SDM)	Print Office	Trustworthy
Return Codes Control Components (CCRs) and Mixing Control Components (CCMs)	Control Components	Trustworthy only as the whole. At least one is honest.
Auditors	Auditors	At least one is trustworthy
Verifier	Auditor's technical aid	At least one honest auditor has a trustworthy aid

Table 1 - Mapping between components in the protocol and in the VEleS Swiss regulation

Table 1 maps the solution modules with those in the VEleS Swiss regulation. This table also shows the trust assumptions defined by the VEleS regulation that must be ensured by the equivalent module in the proposed voting system. These trust assumptions are considered in the design of the voting system

and components, which are trustworthy and designed to be deployed in isolated environments (e.g., using the SDM offline functionalities).

3 System configuration process

During the system configuration process, all the information unrelated to a specific Election Event will be created. The following schema defines the certificate hierarchy of the system configuration.

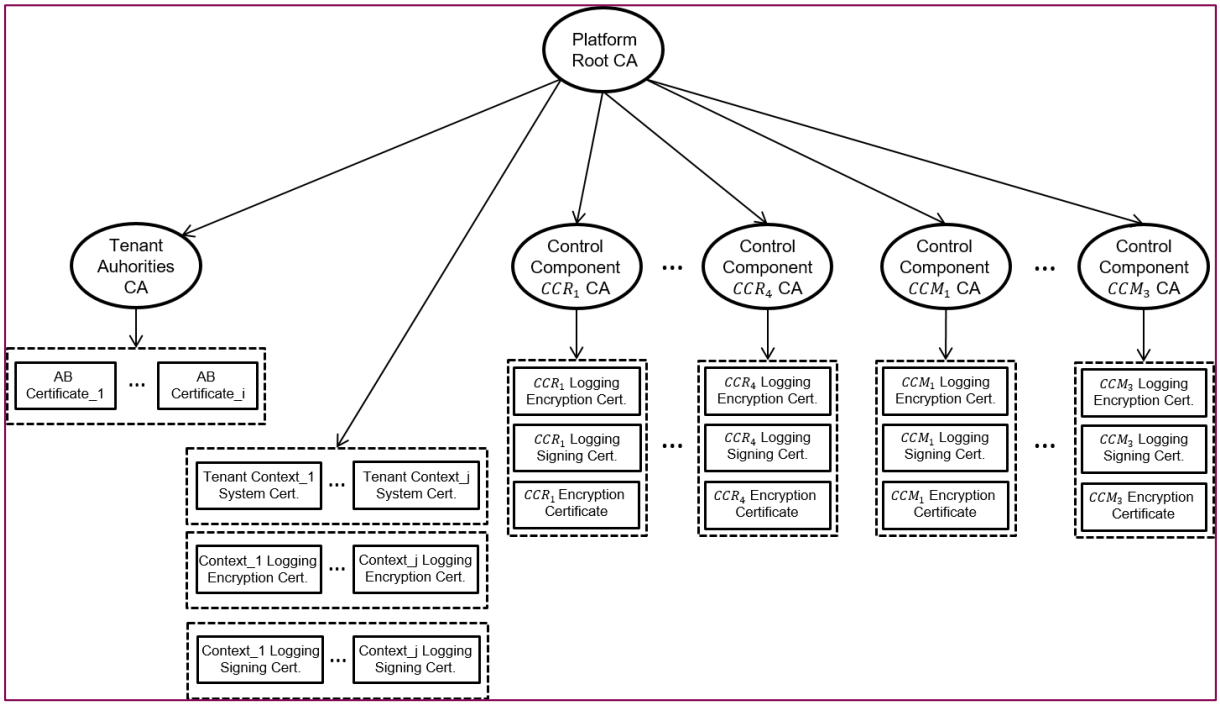

Figure 3 - System certificate hierarchy

First, a Platform Root CA is created. This Platform Root generates all the Tenant Certificates for those Tenants that want to run an election and issues the Tenant CA Certificate.

For each Tenant and each Context of the electronic voting system, the following certificates are generated:

- System certificates
- Logging signing certificates
- Logging encryption certificates

The system key pair is used to encrypt/decrypt the election KeyStore passwords for each context.

The logging signing and encryption key pairs are needed to run the Secure Logs application.

Once a Tenant is registered, it can certify Administration Boards which can digitally sign valid configurations and election results.

The first approach is to generate all the Tenant and Platform configuration information using two command line tools known as Customer Administrator Tools (CATs). These tools allow the creation of credentials for platforms and tenants, as well as the system credentials for an election. Additionally, the CATs allow installing the configuration on the services that will need it to run an election.

In addition, the Platform Root also issues certificates for the Control Components. For each one of them $(CCR_1, CCR_2, CCR_3, CCR_4, CCM_1, CCM_2, CCM_3)$ it generates a Control Component CA that issues the following certificates:

- Encryption certificate
- Logging signing certificate
- Logging encryption certificate

The encryption key pair is used to encrypt/decrypt the passwords of the KeyStores that contain the election private keys, and the logging signing and encryption key pairs needed to run the SecureLog application.

Note: These keys are unique per Control Component (CC) but shared among all Tenants certified by the Platform Root.

Key	Variable	Owner	Meaning
Platform Root CA private key	$PRCA_{sk}$	Platform	This RSA private key is used to issue the Tenant CA, the Control Component CCR_j CAs, the Control Component CCM_j CAs, the CCR_j Logging encryption, the CCR_j Logging signing, the CCM_j Logging encryption, the CCM_j Logging signing, the CCR_j encryption, the CCM_j encryption, the Tenant Context System, the Context Logging Encryption and the Context Logging Signing certificates.
Platform Root CA public key	$PRCA_{pk}$	Platform	This RSA public key is used to verify the Tenant CA, the Control Component CCR_j CAs, the Control Component CCM_j CAs, the CCR_j Logging encryption, the CCR_j Logging signing, the CCM_j Logging encryption, the CCM_j Logging signing, the CCR_j encryption, the CCM_j encryption, the Tenant Context System, the Context Logging Encryption and the Context Logging Signing certificates.

Key	Variable	Owner	Meaning
Tenant CA private key	TCA_{sk}	Tenant	This RSA private key is used to issue the Administration Board certificate.
Tenant CA public key	TCA_{pk}	Tenant	This RSA public key is used to validate the Administration Board certificate.
CCR_j CA private key	$CCRCA_{sk}^{j}$	CCR_j	This RSA private key is used to issue the CCR_j signing certificate.
CCR_j CA public key	$CCRCA_{pk}^{j}$	CCR_j	This RSA public key is used to validate the CCR_j signing certificate.
CCR_j Logging Encryption private key	$CCRlog_{sk}^{j,e}$	CCR_j	This RSA private key is used to decrypt the symmetric keys that are used to compute the CCR_j Secure Log checkpoints.
CCR_j Logging Encryption public key	$CCRlog_{pk}^{j,e}$	CCR_j	This RSA public key is used to encrypt the symmetric keys that are used to compute the CCR_j Secure Log checkpoints.
CCR_j Logging Signing private key	$CCRlog_{sk}^{j,s}$	CCR_j	This RSA private key is used to sign the CCR_j Secure Logs checkpoints.
CCR_j Logging Signing public key	$CCRlog_{pk}^{j,s}$	CCR_j	This RSA public key is used to verify the CCR_j Secure Logs checkpoints signatures.
CCR_j Encryption private key	$CCRe_{sk}^{j}$	CCR_j	This RSA private key is used to decrypt the CCR_j Choice Return Code encryption private key (sk_{CCR_j}), the CCR_j Choice Return Code generation private key (k'_j) and the CCR_j signing private key ($sk_{CCR_j}^{s}$).
CCR_j Encryption public key	$CCRe_{pk}^{j}$	CCR_j	This RSA public key is used to encrypt the CCR_j Choice Return Code encryption private key (sk_{CCR_j}), the CCR_j Choice Return Code generation private key (k'_j) and the CCR_j signing private key ($sk_{CCR_j}^{s}$).
CCM_j CA private key	$CCMCA_{sk}^{j}$	CCM_j	This RSA private key is used to issue the CCM_j signing certificate.
CCM_j CA public key	$CCMCA_{pk}^{j}$	CCM_j	This RSA public key is used to validate the CCM_j signing certificate.

Key	Variable	Owner	Meaning
CCM_j Logging Encryption private key	$CCMlog_{sk}^{j,e}$	CCM_j	This RSA private key is used to decrypt the symmetric keys that are used to compute the CCM_j Secure Log checkpoints.
CCM_j Logging Encryption public key	$CCMlog_{pk}^{j,e}$	CCM_j	This RSA public key is used to encrypt the symmetric keys that are used to compute the CCM_j Secure Log checkpoints.
CCM_j Logging Signing private key	$CCMlog_{sk}^{j,s}$	CCM_j	This RSA private key is used to sign the CCM_j Secure Logs checkpoints.
CCM_j Logging Signing public key	$CCMlog_{pk}^{j,s}$	CCM_j	This RSA public key is used to verify the CCM_j Secure Logs checkpoints signatures.
CCM_j Encryption private key	$CCMe_{sk}^{j}$	CCM_j	This RSA private key is used to decrypt the CCM_j Mixing private key (x_j) and the CCM_j signing private key ($sk_{CCM_j}^{s}$).
CCM_j Encryption public key	$CCMe_{pk}^{j}$	CCM_j	This RSA public key is used to encrypt the CCM_j Mixing private key (x_j) and the CCM_j signing private key ($sk_{CCM_j}^{s}$).
Administration Board private key	AB_{sk}	Administration Board	This RSA private key is used to sign the election configuration and the counting results.
Administration Board public key	AB_{pk}	Administration Board	This RSA public key is used validate the election configuration signatures and the counting results signature.
Tenant Authentication Context System private key	TAC_{sk}	Authentication Context	This RSA private key is used to decrypt the Authentication Token Signer Password.
Tenant Authentication Context System public key	TAC_{pk}	Authentication Context	This RSA public key is used to encrypt the Authentication Token Signer Password.
Tenant Vote Verification Context System private key	TVV_{sk}	Vote Verification Context	This RSA private key is used to decrypt the Choice Return Codes Encryption KeyStore password and the Codes Secret key KeyStore password.

Key	Variable	Owner	Meaning
Tenant Vote Verification Context System public key	TVV_{pk}	Vote Verification Context	This RSA public key is used to decrypt the Choice Return Codes Encryption KeyStore password and the Codes Secret key KeyStore password.
Tenant Election Information Context System private key	TEI_{sk}	Election Information Context	This RSA private key is used to decrypt the Ballot Box KeyStore password and the Election Information Signing KeyStore password.
Tenant Election Information Context System public key	TEI_{pk}	Election Information Context	This RSA public key is used to encrypt the Ballot Box KeyStore password and the Election Information Signing KeyStore password.
Authentication Context Logging Encryption private key	$AClog^{e}_{sk}$	Authentication Context	This RSA private key is used to decrypt the symmetric keys that are used to compute the Authentication Context Secure Log checkpoints.
Authentication Context Logging Encryption public key	$AClog^{e}_{pk}$	Authentication Context	This RSA public key is used to encrypt the symmetric keys that are used to compute the Authentication Context Secure Log checkpoints.
Authentication Context Logging Signing private key	$AClog^{s}_{sk}$	Authentication Context	This RSA private key is used to sign the Authentication Context Secure Logs checkpoints.
Authentication Context Logging Signing public key	$AClog^{s}_{pk}$	Authentication Context	This RSA public key is used to verify the Authentication Context Context Secure Logs checkpoints signatures.
Voting Workflow Context Logging Encryption private key	$VWlog^{e}_{sk}$	Voting Workflow Context	This RSA private key is used to decrypt the symmetric keys that are used to compute the Voting Workflow Context Secure Log checkpoints.
Voting Workflow Context Logging Encryption public key	$VWlog^{e}_{pk}$	Voting Workflow Context	This RSA public key is used to encrypt the symmetric keys that are used to compute the Voting Workflow Context Secure Log checkpoints.

Key	Variable	Owner	Meaning
Voting Workflow Context Logging Signing private key	$VWlog^{s}_{sk}$	Voting Workflow Context	This RSA private key is used to sign the Voting Workflow Context Secure Logs checkpoints.
Voting Workflow Context Logging Signing public key	$VWlog^{s}_{pk}$	Voting Workflow Context	This RSA public key is used to verify the Voting Workflow Context Secure Logs checkpoints signatures.
Vote Verification Context Logging Encryption private key	$VVlog^{e}_{sk}$	Vote Verification Context	This RSA private key is used to decrypt the symmetric keys that are used to compute the Vote Verification Context Secure Log checkpoints.
Vote Verification Context Logging Encryption public key	$VVlog^{e}_{pk}$	Vote Verification Context	This RSA public key is used to encrypt the symmetric keys that are used to compute the Vote Verification Context Secure Log checkpoints.
Vote Verification Context Logging Signing private key	$VVlog^{s}_{sk}$	Vote Verification Context	This RSA private key is used to sign the Vote Verification Context Secure Logs checkpoints.
Vote Verification Context Logging Signing public key	$VVlog^{s}_{pk}$	Vote Verification Context	This RSA public key is used to verify the Vote Verification Context Secure Logs checkpoints signatures.
Voter Material Context Logging Encryption private key	$VMlog^{e}_{sk}$	Vote Material Context	This RSA private key is used to decrypt the symmetric keys that are used to compute the Voter Material Context Secure Log checkpoints.
Voter Material Context Logging Encryption public key	$VMlog^{e}_{pk}$	Vote Material Context	This RSA public key is used to encrypt the symmetric keys that are used to compute the Voter Material Context Secure Log checkpoints.
Voter Material Context Logging Signing private key	$VMlog^{s}_{sk}$	Vote Material Context	This RSA private key is used to sign the Voter Material Context Secure Logs checkpoints.
Voter Material Context Logging Signing public key	$VMlog^{s}_{pk}$	Vote Material Context	This RSA public key is used to verify the Voter Material Context Secure Logs checkpoints signatures.

Key	Variable	Owner	Meaning
Election Information Context Logging Encryption private key	$EIlog_{sk}^{e}$	Election Information Context	This RSA private key is used to decrypt the symmetric keys that are used to compute the Election Information Context Secure Log checkpoints.
Election Information Context Logging Encryption public key	$EIlog_{pk}^{e}$	Election Information Context	This RSA public key is used to encrypt the symmetric keys that are used to compute the Election Information Context Secure Log checkpoints.
Election Information Context Logging Signing private key	$EIlog_{sk}^{s}$	Election Information Context	This RSA private key is used to sign the Election Information Context Secure Logs checkpoints.
Election Information Context Logging Signing public key	$EIlog_{pk}^{s}$	Election Information Context	This RSA public key is used to verify the Election Information Context Secure Logs checkpoints signatures.
Certificate Registry Context Logging Encryption private key	$CRlog_{sk}^{e}$	Certificate Registry	This RSA private key is used to decrypt the symmetric keys that are used to compute the Certificate Registry Secure Log checkpoints.
Certificate Registry Context Logging Encryption public key	$CRlog_{pk}^{e}$	Certificate Registry	This RSA public key is used to encrypt the symmetric keys that are used to compute the Certificate Registry Secure Log checkpoints.
Certificate Registry Context Logging Signing private key	$CRlog_{sk}^{s}$	Certificate Registry	This RSA private key is used to sign the Certificate Registry Secure Logs checkpoints.
Certificate Registry Context Logging Signing public key	$CRlog_{pk}^{s}$	Certificate Registry	This RSA public key is used to verify the Certificate Registry Secure Logs checkpoints signatures.

Key	Variable	Owner	Meaning
Election Information Signing private key	EI_{sk}^{s}	Election Information Context	This RSA private key is used to sign the cleansed ballot box (the input of the first mixing Control Component).
Election Information Signing public key	EI_{pk}^{s}	Election Information Context	This RSA public key is used to verify the signature of the cleansed ballot box (the input of the first mixing Control Component).

Table 2 - System keys notation

3.1 Platform Root constitution and registration

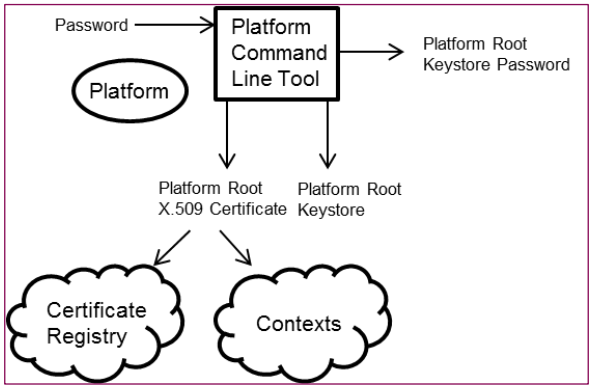

Figure 4 - Platform Root constitution and registration

The Platform Command Line Tool is used to generate the Platform Root Credentials.

- Call the RSA Key pair generation primitive and obtain the Platform Root key pair.
- The Platform user is asked to introduce a KeyStore password.
- Call the X509 certificate generation primitive to generate the Platform Root certificate, self-signed with the private RSA key. The certificate contains the CA name, the validity period and the certificate name field values.
- The private RSA key is stored into a KeyStore and seal it with the password.

The Platform Root certificate is uploaded to the Certificate Registry and to the contexts. The KeyStore password is kept by the Platform user to be used to issue tenant and system certificates.

3.2 Tenant constitution and registration

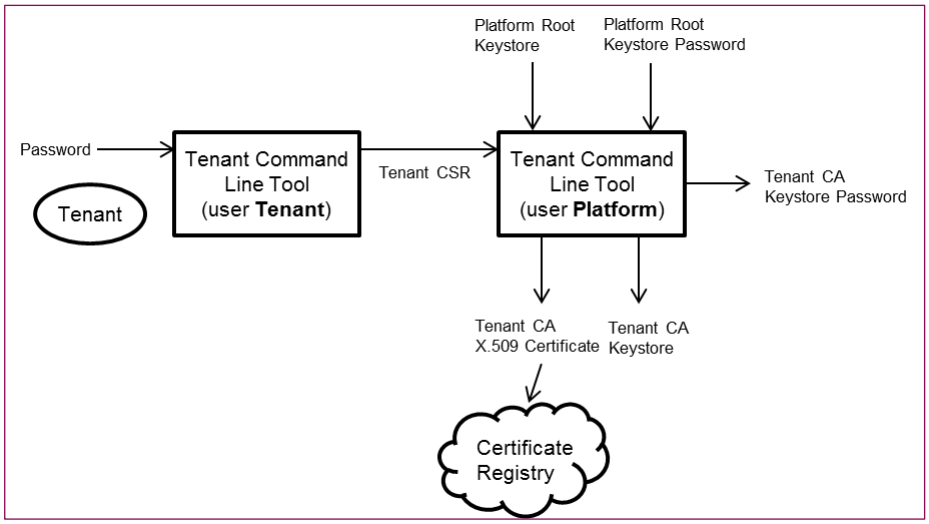

Figure 5 - Tenant constitution and registration

3.2.1 Tenant constitution

The Tenant Command Line Tool is used by the Tenant user to generate their credentials:

- Call the RSA Key pair generation primitive and obtain the Tenant key pair
- The Tenant user is asked to introduce a KeyStore password.
- A CSR with the public key is generated and sent to the Platform Root (for instance, by authenticate mail). The CSR contains the tenant identifier in the common name. The validity period should be configurable.
- The private RSA key is stored into a KeyStore and seal it with the password.
- The KeyStore and password are kept by the Tenant locally.

3.2.2 Tenant registration

Precondition: The platform host has a CA key pair, constituted by a KeyStore containing the private key, and a self-signed certificate (containing the public key), which has been installed in the contexts.

The platform host registers the Tenant by issuing the X.509 for the Tenant CA, from the CSR created in the previous step.

- The Command Line Tool requires entering the Platform Root KeyStore password to retrieve the Platform Root private signing key.
- The Platform Host CA calls the X509 certificate generation primitive to generate the Tenant CA X509 certificate from the existing CSR and using the Platform Host CA private key.

- The Tenant CA certificate is uploaded only to the Certificate Registry. The registry checks that the Tenant CA certificate has been issued by the Platform Host CA.

When the contexts need the Tenant Authorities CA Certificate, they request it from the Certificate Registry.

3.3 System context credentials

Precondition: The platform host has a CA key pair, constituted by a KeyStore containing the private key and a certificate, which has been installed in the contexts.

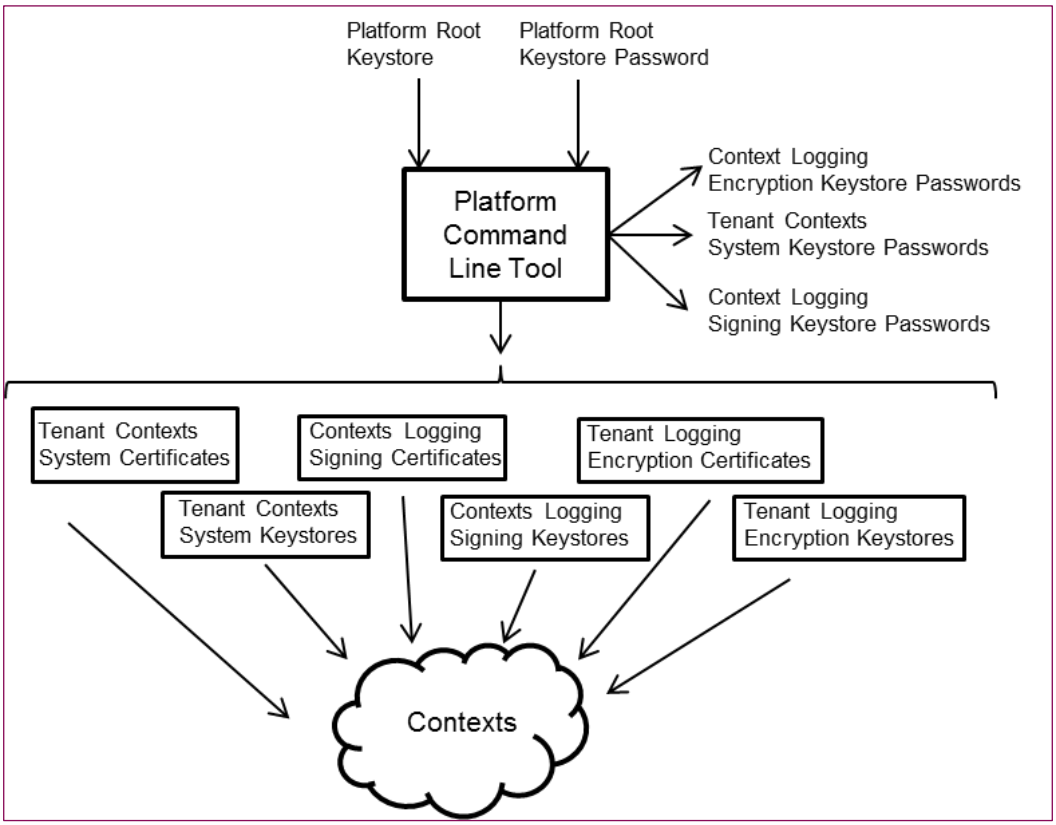

Figure 6 - System Context credentials generation

3.3.1 Logging Context Keys

The platform host generates two key pairs for each context to start logging information in a secure way. One key pair will be used to encrypt and the other to sign.

- Call twice to the RSA Key pair generation primitive and obtain the Logging Context key pairs
  - Authentication Context: $(AClog^{e}_{pk}, AClog^{e}_{sk}), (AClog^{s}_{pk}, AClog^{s}_{sk})$
  - Voting Workflow Context: $(VWlog^{e}_{pk}, VWlog^{e}_{sk}), (VWlog^{s}_{pk}, VWlog^{s}_{sk})$
  - Voter Material Context: $(VMlog^{e}_{pk}, VMlog^{e}_{sk}), (VMlog^{s}_{pk}, VMlog^{s}_{sk})$
  - Election Information Context: $(EIlog^{e}_{pk}, EIlog^{e}_{sk}), (EIlog^{s}_{pk}, EIlog^{s}_{sk})$

  - Vote Verification Context: $(VVlog^{e}_{pk}, VVlog^{e}_{sk}), (VVlog^{s}_{pk}, VVlog^{s}_{sk})$
  - Certificate Registry: $(CRlog^{e}_{pk}, CRlog^{e}_{sk}), (CRlog^{s}_{pk}, CRlog^{s}_{sk})$

- The Command Line Tool requires entering the Platform Root KeyStore password to retrieve the Platform Root private signing key $(PRCA_{sk})$.
- Call twice to the X509 certificate generation primitive with the Platform Root CA private key $(PRCA_{sk})$ to Logging Context Signing and Encryption X509 certificates. The certificates should contain in the "common name", the name of the context for which they are issued. This is the list of contexts:
  - Authentication Context
  - Election Information Context
  - Vote Verification Context
  - Voting Workflow Context
  - Voter Material Context
  - Certificate Registry
- The Context Logging signing/encryption certificates and KeyStores are uploaded to the respective contexts.
- The Context Logging signing/encryption KeyStore passwords are kept by the user and never stored on disk.
- Each context verifies that the certificates have been issued by the Platform Host CA

3.3.2 Context System Keys

The platform host generates the Tenant Contexts System key pairs, passwords and KeyStores:

- Call the RSA Key pair generation primitive and obtain the Tenant Context System key pair: $(TAC_{pk}, TAC_{sk}), (TVV_{pk}, TVV_{sk}), (TEI_{pk}, TEI_{sk})$.
- The Command Line Tool requires entering the platform root KeyStore password to retrieve the Platform Root CA private key $(PRCA_{sk})$.
- Call the X509 certificate generation primitive with the Platform Root CA private key $(PRCA_{sk})$ to generate the Tenant Context System X509 Certificates. Certificates and KeyStores are uploaded to the respective context.
- The Tenant System Context KeyStore passwords are kept by the user and never stored on disk

The system keys are used to encrypt/decrypt the KeyStore passwords of the election KeyStores

Additionally, for the Election Information Context, an extra pair of keys (**Election Information Signing Key Pair**) are generated to sign the cleansed Ballot Box after the election period ends and before sending it to the mixing process. To generate this pair of keys, the following steps are executed per each Tenant:

- Call the RSA Key pair generation primitive and obtain the Election Information Signing key pair (EI^S_{pk}, EI^S_{sk}).
- The Command Line Tool requires entering the platform root KeyStore password to retrieve the Platform Root CA private key $(PRCA_{sk})$.
- Call the X509 certificate generation primitive with the Platform Root CA private key $(PRCA_{sk})$ to generate the Election Information Signing Certificate. The common name will contain the Tenant ID and the Service ID.
- The Election Information Signing KeyStore password is encrypted with the Tenant Election Information Context system public key (TEI_{pk}).
- The Election Information Signing Certificate, the KeyStore and the encrypted password are uploaded to the Election Information Context.

3.4 Administration Board constitution and registration

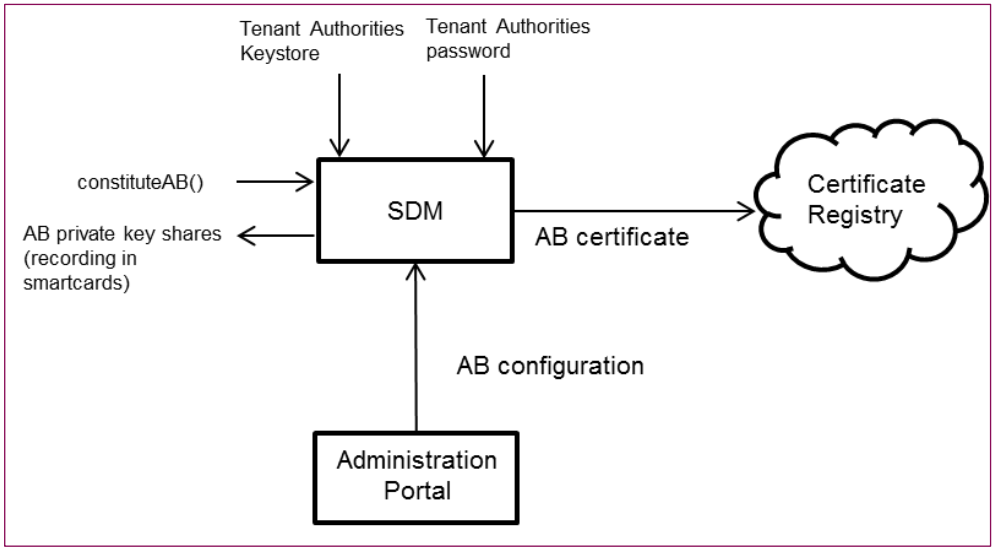

Figure 7 - Administration Board constitution and registration

3.4.1 Administration Board constitution

The Administration Board information such as Member details, number of shares and threshold; is configured in the Administration Portal (AP). The Administration Board is then locally constituted in the Print Office using the offline SDM through these steps:

- The Administration Board members (for a specific tenant) are configured in the AP. Several Administration Boards can be configured and be constituted for the same tenant. This configuration is synchronized to the Print Office environment.
- Call the RSA Key pair generation primitive and obtain the Administration Board key pair (AB_{pk}, AB_{sk}). The private key splitting functionality receives as input the number of shares, the threshold and the private key. The private key is divided into shares and each share is stored in a PIN-protected smartcard:
    - Smartcards are initialized (fabric configuration or previous shares are erased).
    - The RSA private key is divided into as many shares as provided by the configuration, and with the configured threshold calling the Shamir Threshold Secret Sharing split algorithm.
    - Each share is digitally signed with the Administration Board private key (AB_{sk}).
    - Each signed share is written in a PIN-protected smartcard. Each Administration Board member sets their PIN.
    - The Administration Board public key (AB_{pk}) is stored in a CSR (certificate signing request), which contains an identifier for the Administration Board in the common name.

3.4.2 Administration Board registration

An Administration Board is certified by a Tenant, so that it can sign election configurations and results which are held in the scope of that Tenant. The certification of the Administration Board is similar to the certification of the Tenant:

- The Command Line Tool requires entering the Tenant CA KeyStore password to retrieve the Tenant CA private key (TCA_{sk}).
- Call the X509 certificate generation primitive with the Tenant CA private key (TCA_{sk}) to issue the Administration Board X509 certificate.
    - The validity period of the certificate should be configurable
    - The certificate type is “Signing” and “Non-Repudiation”
- The Administration Board certificate is uploaded to the Certificate Registry.

3.5 Control Components Credentials

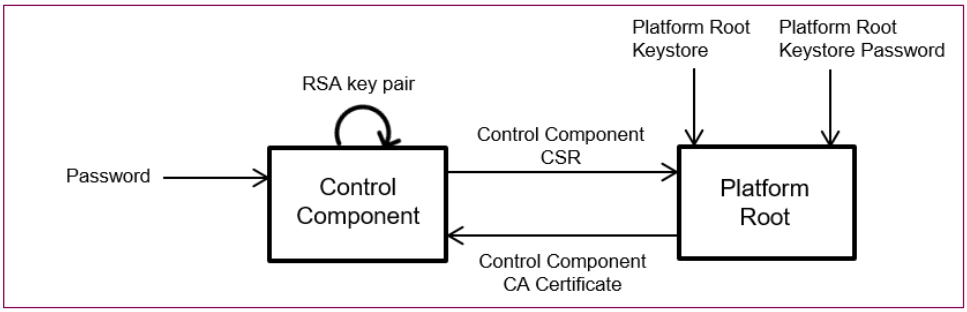

Figure 8 - Control Component Constitution and Registration

3.5.1 Control Component CA

The platform host registers the Control Component by issuing the X.509 for the Control Component CA. For each Choice Return Codes Control Component and each Mixing Control Component:

- Call the RSA Key pair generation primitive and obtain the $(CCRCA_{pk}^{j}, CCRCA_{sk}^{j})/(CCMCA_{pk}^{j}, CCMCA_{sk}^{j})$. The Command Line Tool requires entering the platform root KeyStore password to retrieve the Platform Root CA private key $(PRCA_{sk})$.
- The Platform Host CA calls the X509 certificate generation primitive using its private key, to generate the Control Component CA X509 certificate.
- The Control Component CA certificate and KeyStores are uploaded to the respective CC.
- Each Control Component checks the certificate chain [Control Component CA, Platform Root CA].
- The KeyStore and password are kept by the Control Component, locally.

3.5.2 Control Components Logging Keys

Each Control Component generates two key pairs to start logging information in a secure way. One key pair will be used to encrypt and the other to sign.

- Calls twice to the RSA Key pair generation primitive and obtain the Control Components Logging key pairs: $\left(CCRlog_{pk}^{j,e}, CCRlog_{sk}^{j,e}\right), \left(CCRlog_{pk}^{j,s}, CCRlog_{sk}^{j,s}\right), \left(CCMlog_{pk}^{j,e}, CCMlog_{sk}^{j,e}\right), \left(CCMlog_{pk}^{j,s}, CCMlog_{sk}^{j,s}\right)$.
- The Control Component retrieves the Control Component CA private key $(CCRCA_{sk}^{j}/CCMCA_{sk}^{j})$ stored in the KeyStore.
- Call twice to the X509 certificate generation primitive with the Control Component CA private key $(CCRCA_{sk}^{j}/CCMCA_{sk}^{j})$ to generate the logging signing and encryption X509 certificates.

These certificates should contain in the common name, the name of the component for which they are issued.

- The logging signing/encryption private keys are stored in a KeyStore.

3.5.3 Control Component Encryption Keys

The Control Components encryption key will be generated through the following steps:

- Call the RSA Key pair generation primitive and obtain the Control Component Encryption key pair.
- The Control Component retrieves the Control Component CA private key $(CCRCA^{j}_{sk}/CCMCA^{j}_{sk})$ stored in the KeyStore.
- Call the X509 certificate generation primitive with the Control Component CA private key $(CCRCA^{j}_{sk}/CCMCA^{j}_{sk})$ to generate the Control Component Encryption X509 Certificate. This certificate should contain in the common name, the name of the component for which they are issued.
- The private RSA key is stored into a KeyStore and seal it with the password.

4 Election configuration process

The following diagram is an overview of the election configuration process. This process is mainly managed by the Voting Card Generation and Election Keys Generation modules in the Print Office component. Furthermore, the modules of the Print Office components need to interact with the Control Components through the Election Information Context module of the Voting Server component.

Since Voting Card Generation, Election Keys Generation and Election Information Context modules uses the Secure Data Manager software component for performing these operations, for simplicity reasons, the explanation refers to the Secure Data Manager instead of specifying the Print Office and Voting Server modules. The differentiation between offline Secure Data Manager when talking about the Print Office modules, and online Secure Data Manager when talking about the Voting Server module (that mainly acts as a proxy) is maintained though. Sections 4.8 to 4.11 are omitted in the diagram since they are just a description of how the information is grouped.

The Election Configuration process begins in the Administration Portal, where the elements needed to run an election can be configured. These elements are:

- Ballots
- Ballot Boxes
- Electoral Authority Lists
- Voting Card Sets

In the Administration Portal, no cryptographic operations are done, all the cryptographic information needed to run an election is generated either in the Print Office environment executing the offline Secure Data Manager, or in the Control Components.

Once the previously described elements are fully configured, they are downloaded from the Print Office or uploaded to the Control Components.

It is mandatory for an Election Event to have a constituted Administration Board assigned to it, to start the process.

The configuration generated in the Control Components is downloaded to the Print Office to be signed by the Administration Board as part of the election configuration.

The processes detailed in this section are all executed in the Print Office environment using the offline Secure Data Manager, except for those where it is explicitly indicated that they are run in the Control Components.

Before describing the Election configuration process in detail, some notations regarding keys, codes and variables are needed.

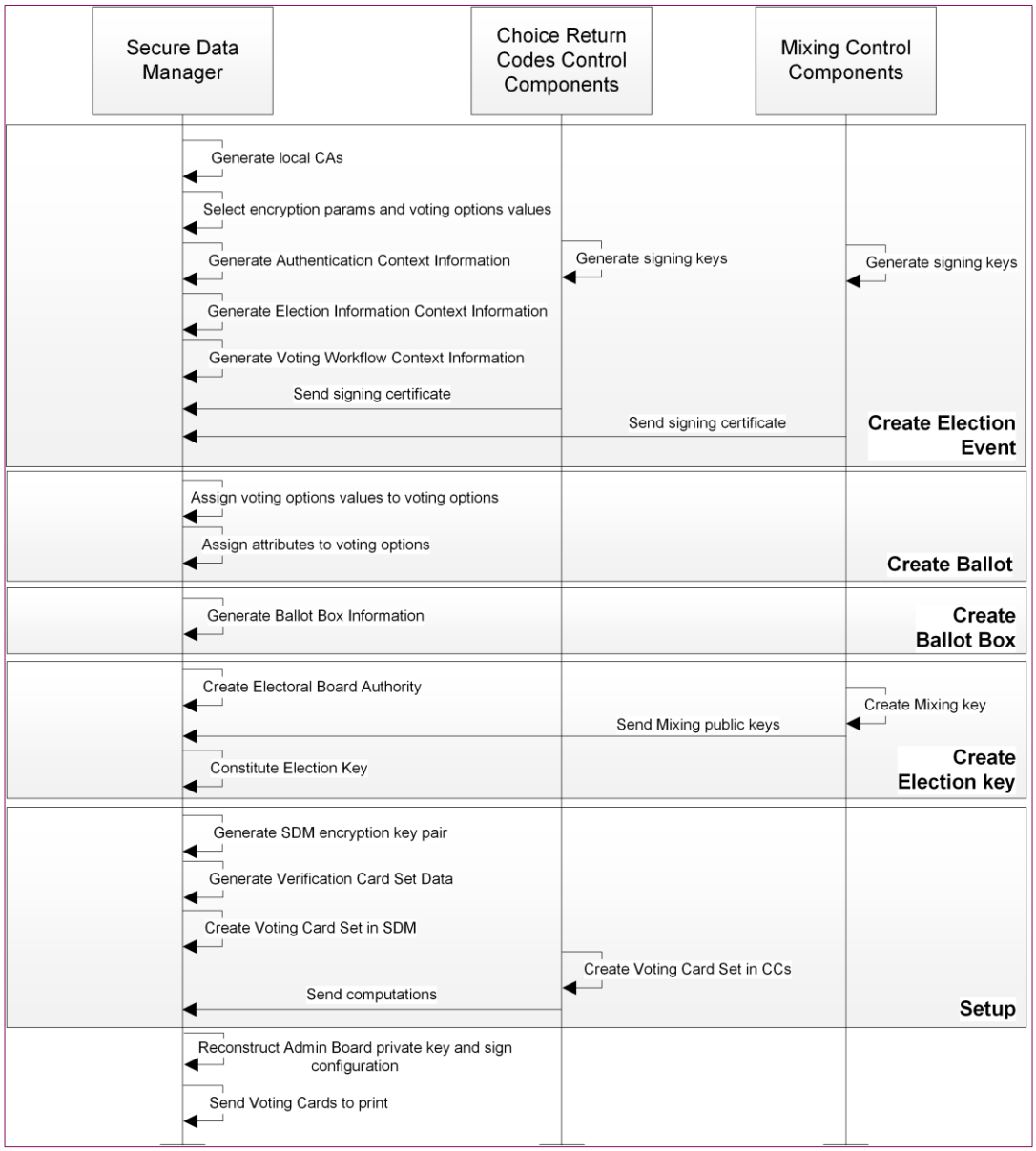

Figure 9 - Election configuration phase overview

4.1 Notation

Variable	Meaning
p	Defines the order of the group.
q	Defines the order of the subgroup of quadratic residues.
g	The generator of the mathematical group.
j	Index used to refer to a specific Control Component
i	Index used to refer to a specific voting options or specific Choice Return Code.

Variable	Meaning
ψ	Maximum number of options a voter can select (it is also the number of elements of the Choice Return Codes Encryption private and public keys).
id	Index used to refer to a specific voter.
n	Number of voting options available in the election
k	Number of allowed write-ins
v_i	The encoding (a prime number) of the voting option selected by the voter.
m	Number of elements of the Election key, that is the number of write-ins k plus 1.

Table 3 - Variables notation

IDs	Meaning
vcd_{id}	Voting Card ID. It identifies the Voting Card Data corresponding to a specific voter.
$vcds_{id}$	Voting Card Set ID. It identifies the set of information common to a group of Voting Card IDs. It has one to one correspondence with the Verification Card Set ID.
vc_{id}	Verification Card ID. It identifies the Verification Card Data corresponding to a specific voter.
vcs_{id}	Verification Card Set ID. It identifies the set of information common to a group of Verification Card IDs. It has one to one correspondence with the Voting Card Set ID.
c_{id}	Credential ID corresponding to a specific voter.

Table 4 - Voter Identifiers

IDs	Meaning
$bbid$	Ballot box ID. It identifies a ballot box.
bid	Ballot ID. It identifies a ballot. One ballot can be related to more than one Ballot Boxes.
$eeid$	Election Event ID. It identifies an election.

Table 5 - Election Identifiers

Codes	Meaning
SVK_{id}	Start Voting Key associated to id.
CC_i^{id}	i-th short Choice Return Code associated to id.
pCC_i^{id}	i-th partial Choice Return Code associated to id.
lCC_i^{id}	i-th long Choice Return Code associated to id.
pC_i^{id}	i-th pre-Choice Return Code associated to id.
VCC^{id}	short Vote Cast Return Code associated to id.
$lVCC^{id}$	long Vote Cast Return Code associated to id.
$pVCC^{id}$	pre-Vote Cast Return Code associated to id.
BCK^{id}	Ballot Casting Key associated to id.
CM^{id}	Confirmation Message associated to id.

Table 6 - Codes notation

Key	Variable	Occurrence	Owner	Meaning
Election Event Root CA private key	$EECA_{sk}$	1 per Election Event	Tenant	This RSA private key is used to issue the Services CA, Authorities CA and Credentials CA certificates.
Election Event Root CA public key	$EECA_{pk}$	1 per Election Event	Tenant	This RSA public key is used to validate the Services CA, Authorities CA and Credentials CA certificates.
Services CA private key	SCA_{sk}	1 per Election Event	Tenant	This RSA private key is used to issue the Authentication Token Signer, the Ballot Box, the Verification Card Set Issuer and the Vote Cast Code Signer certificates.
Services CA public key	SCA_{pk}	1 per Election Event	Tenant	This RSA public key is used to validate the Authentication Token Signer, the Ballot Box, the Verification Card Set Issuer and the Vote Cast Code Signer certificates.
Authorities CA private key	ACA_{sk}	1 per Election Event	Tenant	This RSA private key is used to sign the Electoral Board private key (EB_{sk}) shares.
Authorities CA public key	ACA_{pk}	1 per Election Event	Tenant	This RSA public key is used to validate the signature of the Electoral Board private key (EB_{sk}) shares.
Credentials CA private key	CCA_{sk}	1 per Election Event	Tenant	This RSA private key is used to issue the voters' certificates.
Credentials CA public key	CCA_{pk}	1 per Election Event	Tenant	This RSA public key is used to validate the voters' certificates.

Table 7 - Election CA keys notation

Key	Variable	Occurrence	Owner	Meaning
Authentication Token Signer private key	ATs_{sk}	1 per Election Event	Voting Server (Authentication Context)	This RSA private key is used both for signing the Authentication Token and the Server Challenge.
Authentication Token Signer public key	ATs_{pk}	1 per Election Event	Voting Server (Authentication Context)	This RSA public key is used both for verifying the Authentication Token signature and the Server Challenge signature.
Ballot Box Signer private key	$BBs_{sk}^{bb_{id}}$	1 per Ballot Box	Voting Server (Election Information Context)	This RSA private key is used to sign the receipt and the datasets generated during the Ballot Box Export.
Ballot Box Signer public key	$BBs_{pk}^{bb_{id}}$	1 per Ballot Box	Voting Server (Election Information Context)	This RSA public key is used to validate the signature generated with the corresponding private key.
Verification Card private key	k_{id}	1 per voter	Voter	This ElGamal private key is used to compute the partial Choice Return Code (pCC_i^{id}) and the confirmation message (CM^{id}).
Verification Card public key	K_{id}	1 per voter	Voter	This ElGamal public key is used to generate and verify the Exponentiation Proof computed by the voting client.
Verification Card Set Issuer private key	VCI_{sk}	1 per Verification Card Set	Secure Data Manager	This RSA private key is used to sign all the Verification Card Public keys (K_{id}) corresponding to the Verification Card IDs belonging to the Verification Card Set.
Verification Card Set Issuer public key	VCI_{pk}	1 per Verification Card Set	Voting Server (Vote Verification Context)	This RSA public key is used to verify the signature generated with the corresponding private key.

Key	Variable	Occurrence	Owner	Meaning
Vote Cast Return Code Signer private key	$VCCs_{sk}$	1 per Verification Card Set	Voting Server (Vote Verification Context)	This RSA private key is used to sign each short Vote Cast Return Code(VCC^{id}) corresponding to each Verification Card ID belonging to the set.
Vote Cast Return Code Signer public key	$VCCs_{pk}$	1 per Verification Card Set	Voting Server (Vote Verification Context)	This RSA public key is used to verify the signature generated with the corresponding private key.
Codes Secret Key	C_{sk}	1 per Verification Card Set	Voting Server (Vote Verification Context)	This symmetric key is used to compute the long Choice Return Codes (lCC_i^{id}) and the long Vote Cast Return Codes ($lVCC^{id}$).
Credential ID authentication private key	$k_{C_{id}}^{a}$	1 per voter	Voter	This RSA private key is used to sign the Client Challenge.
Credential ID authentication public key	$K_{C_{id}}^{a}$	1 per voter	Voter	This RSA public key is used to verify the signatures generated with the corresponding private key.
Credential ID signing private key	$k_{C_{id}}^{s}$	1 per voter	Voter	This RSA private key is used to sign the vote and the Confirmation Message (CM^{id}).
Credential ID signing public key	$K_{C_{id}}^{s}$	1 per voter	Voter	This RSA public key is used to verify the signatures generated with the corresponding private key.
KeyStore symmetric encryption key	$KSkey_{id}$	1 per voter	Voter	This symmetric key is used to seal the voter's KeyStore.

Key	Variable	Occurrence	Owner	Meaning
CCR_j Choice Return Codes Encryption private key	sk_{CCR_j}	1 per Verification Card Set	Control Component CCR_j	This ElGamal private key is used to partially decrypt the encrypted pre-Choice Return Codes (pC_i^{id}). This key will have as many elements (ψ) as the maximum number of options a voter can select. We will refer to one specific element as $sk_{CCR_j}^{(i)}$.
CCR_j Choice Return Codes Encryption public key	pk_{CCR_j}	1 per Verification Card Set	Control Component CCR_j	This ElGamal public key is used during the computation of the Choice Return Codes Encryption public key (pk_{CCR}). This key will have as many elements (ψ) as the maximum number of options a voter can select. We will refer to one specific element as $pk_{CCR_j}^{(i)}$.
CCR_j Choice Return Codes Generation private key	k'_j	1 per Verification Card Set	Control Component CCR_j	This ElGamal private key is used to derived Voter Choice Return Code generation private key (k_{id}^j) and the Voter Vote Cast Return Code generation private key (kc_{id}^j).
CCR_j Choice Return Codes Generation public key	$g^{k'_j}$	1 per Verification Card Set	Control Component CCR_j	This is the ElGamal public key corresponding to the CCR_j Choice Return Codes Generation private key (k'_j).

Key	Variable	Occurrence	Owner	Meaning
Choice Return Codes Encryption private key	sk_{CCR}	1 per Verification Card Set	Control Components	This ElGamal private key is a combination of the CCR_j Choice Return Codes Encryption private keys (sk_{CCR_j}). This key will have as many elements (ψ) as the maximum number of options a voter can select. We will refer to one specific element as $sk_{CCR}^{(i)}$.
Choice Return Codes Encryption public key	pk_{CCR}	1 per Verification Card Set	Control Components	This ElGamal public key is used to encrypt the partial Choice Return Codes (pCC_i^{id}) in the voting client. This key will have as many elements (ψ) as the maximum number of options a voter can select. We will refer to one specific element as $pk_{CCR}^{(i)}$.
Voter Choice Return Code generation private key	k_{id}^j	1 per voter	Control Component CCR_j	This ElGamal private key is used by the Control Component CCR_j to compute the exponentiation of the encrypted partial Choice Return Codes (pCC_i^{id}).
Voter Choice Return Code generation public key	K_{id}^j	1 per voter	Control Component CCR_j	This ElGamal public key is used to compute the exponentiation proof of the encrypted partial Choice Return Codes (pCC_i^{id}) in the Control Component and to verify it later.
Voter Vote Cast Return Code generation private key	kc_{id}^j	1 per voter	Control Component CCR_j	This ElGamal private key is used by the Control Component CCR_j to compute the exponentation of the Confirmation Message (CM^{id}).

Key	Variable	Occurrence	Owner	Meaning
Voter Vote Cast Return Code generation public key	Kc_{id}^{j}	1 per voter	Control Component CCR_j	This ElGamal public key is used to compute the exponentiation proof of the confirmation message (CM^{id}). in the Control Component and to verify it later.
Electoral Board private key	EB_{sk}	1 per Electoral Board	Electoral Board	This ElGamal private key is split and each piece in stored in a smartcard belonging to one Electoral Board member. The reconstructed key is used to perform the final decryption. In case write-ins are enabled, this key will have as many elements as the number of write-ins (k) plus 1.
Electoral Board public key	EB_{pk}	1 per Electoral Board Authority	Electoral Board	This ElGamal public key is used to compute the Election public key (EL_{pk}). In case write-ins are enabled, this key will have as many elements as the number of write-ins (k) plus 1.
CCM_j Mixing private key	x_j	1 per Control Component CCM_j	Control Component CCM_j	This ElGamal private key is used to perform partial decryption in the corresponding CCM_j. In case write-ins are enabled, this key will have as many elements as the number of write-ins (k) plus 1.
CCM_j Mixing public key	g^{x_j}	1 per Control Component CCM_j	Control Component CCM_j	This ElGamal public key is used to compute the Election public key (EL_{pk}). In case write-ins are enabled, this key will have as many elements as the number of write-ins (k) plus 1.

Key	Variable	Occurrence	Owner	Meaning
Election private key	EL_{sk}	1 per Electoral Board Authority	Control Components CCM	This ElGamal private key is a combination of the Electoral Board private key (EB_{sk}) and the Control Components Mixing private keys (x_j). In case write-ins are enabled, this key will have as many elements as the number of write-ins plus 1.
Election public key	EL_{pk}	1 per Electoral Board Authority	Control Components CCM	This ElGamal private key is a combination of the Electoral Board public key (EB_{pk}) and the Control Components Mixing public keys (g^{x_j}) and it is used to encrypt the voting options. In case write-ins are enabled, this key will have as many elements as the number of write-ins plus 1.
CCR_j signing private key	$sk^s_{CCR_j}$	1 per Election Event	Control Component CCR_j	This RSA private key is used to sign the information generated by CCR_j.
CCR_j signing public key	$pk^s_{CCR_j}$	1 per Election Event	Control Component CCR_j	This RSA public key is used to verify signatures generated with the corresponding private key.
CCM_j signing private key	$sk^s_{CCM_j}$	1 per Election Event	Control Component CCM_j	This RSA private key is used to sign the information generated by CCM_j.
CCM_j signing public key	$pk^s_{CCM_j}$	1 per Election Event	Control Component CCM_j	This RSA public key is used to verify signatures generated with the corresponding private key.

Key	Variable	Occurrence	Owner	Meaning
Secure Data Manager private key	sk_{SDM}	1 per Election Event	Secure Data Manager	This ElGamal private key is used by the Print Office to decrypt the prime numbers and the ballot casting key (BCK^{id})once they are exponentiated by the Control Components.
Secure Data Manager public key	pk_{SDM}	1 per Election Event	Secure Data Manager	This ElGamal public key is used by the Print Office to encrypt the prime numbers and the ballot casting key (BCK^{id}) to be sent to the Control Components.

Key	Variable	Occurrence	Owner	Meaning
Choice Return Code encryption symmetric key	$skcc_i^{id}$	1 per Choice Return Code	Voting Server (Vote Verification Context)	This derived key is used to encrypt the short Choice Return Code (CC_i^{id})
Vote Cast Return Code encryption symmetric key	$skvcc^{id}$	1 per Vote Cast Return Code	Voting Server (Vote Verification Context)	This derived key is used to encrypt the short Vote Cast Return Code (VCC^{id}) and its signature.

Table 8 - Election keys notation

4.2 Create Election Event

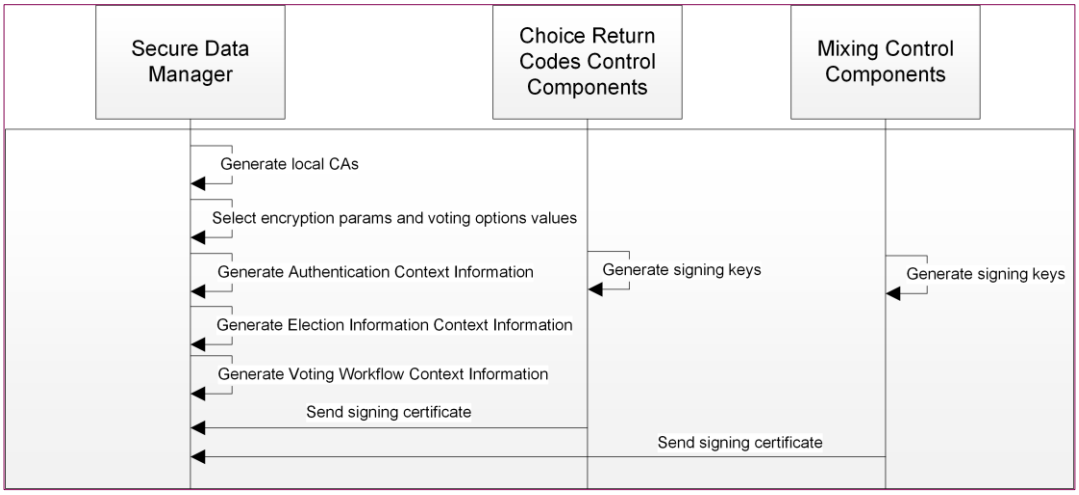

Figure 10 - Create Election Event

This process creates all the information related to an Election Event, which includes the election configuration for the contexts and for the Control Components. A unique Election Event ID ($eeid$) is generated in this step, then the subsequent processes are executed.

The Election Event Certificate hierarchy is the following:

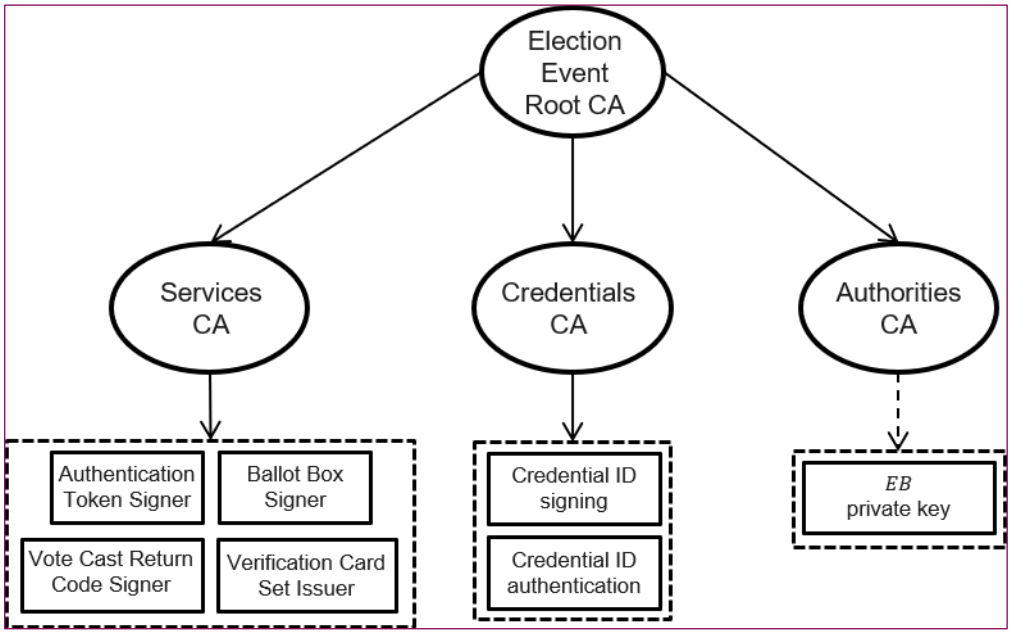

Figure 11 - Election Event certificate hierarchy

In addition to this certificate hierarchy, the Control Components will create a Control Component signing certificate per Election.

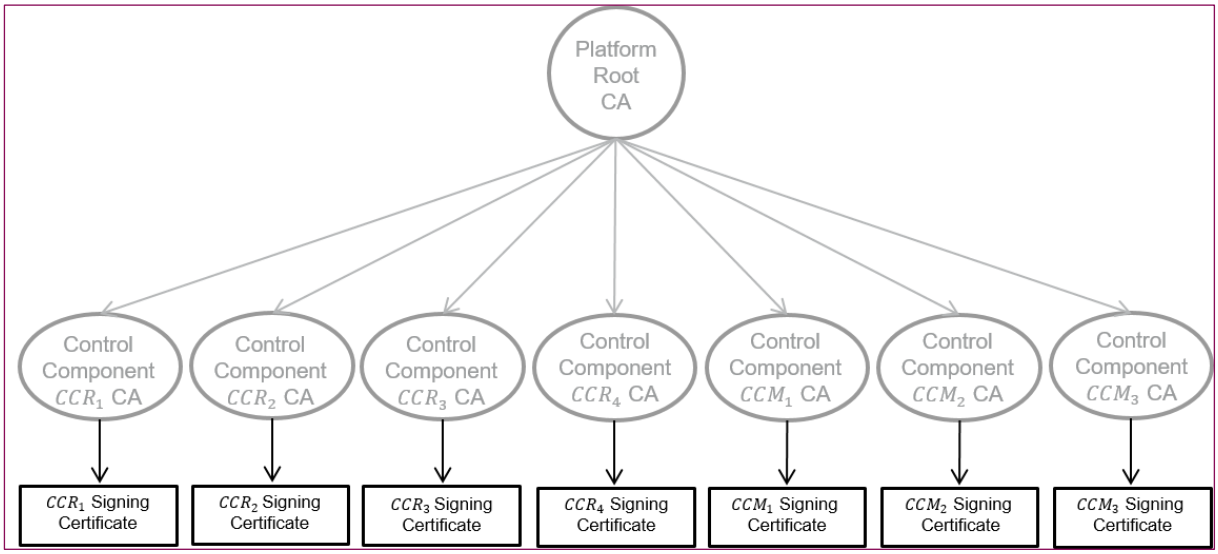

Figure 12 - Control Components Election Event certificate hierarchy

4.2.1 Generation of local Certification Authorities (CA)

The first information to be created during the Election configuration process are the keys and certificates for the Certification Authorities. For each one of them we define below how they should be generated:

1) Election Event Root CA

    1) Call the RSA Key pair generation primitive and obtain the CA key pair:($EECA_{pk}$, $EECA_{sk}$).

2) Call the X509 certificate generation primitive to generate a self-signed certificate using $EECA_{sk}$. The certificate contains the CA name, the Election Event ID $(eeid)$, the validity period and the certificate name field values.
3) Call the Random value generation primitive to generate a random password of length 26 chars in base 32.
4) Store the private RSA key into a KeyStore and seal it with the password.
5) Passwords must be stored encrypted.

2) Services CA
    1) Call the RSA Key pair generation primitive and obtain the CA key pair: (SCA_{pk}, SCA_{sk}).
    2) Call the X509 certificate generation primitive to generate a certificate signed with the $EECA_{sk}$. The certificate contains the CA name, the Election Event ID $(eeid)$, the validity period and the certificate name field values.
    3) Call the Random value generation primitive to generate a random password of length 26 chars in base 32.
    4) Store the private RSA key into a KeyStore and seal it with the password.
    5) Passwords must be stored encrypted.
3) Authorities CA
    1) Call the RSA Key pair generation primitive and obtain the CA key pair: (ACA_{pk}, ACA_{sk}).
    2) Call the X509 certificate generation primitive to generate a certificate signed with the $EECA_{sk}$. The certificate contains the CA name, the Election Event ID $(eeid)$, the validity period and the certificate name field values.
    3) Call the Random value generation primitive to generate a random password of length 26 chars in base 32.
    4) Store the private RSA key into a KeyStore and seal it with the password.
    5) Passwords must be stored encrypted.
4) Credentials CA
    1) Call the RSA Key pair generation primitive and obtain the CA key pair: (CCA_{pk}, CCA_{sk}).
    2) Call the X509 certificate generation primitive to generate a certificate signed with the $EECA_{sk}$. The certificate contains the CA name, the Election Event ID $(eeid)$, the validity period and the certificate name field values.
    3) Call the Random value generation primitive to generate a random password of length 26 chars in base 32.

4) Store the private RSA key into a KeyStore and seal it with the password.

5) Passwords must be stored encrypted.

4.2.2 Selection of encryption parameters and voting option values generation

The encryption parameters (p, q, g) used for the ElGamal encryption scheme satisfy the following conditions:

5) p is a safe prime such that $p = 2q + 1$. q is also prime (2047 bits), and p is of a defined length (2048 bits).

6) The generator g is of order q.

7) The subgroup of $\mathbb{Z}_p^*$ of order q, defines the quadratic residue group over which the encryption scheme is defined.

The values used to represent the voting options in the ballots are chosen from a list of prime numbers in such a way that they fulfil certain conditions regarding the encryption parameters generated (prime numbers should be quadratic residues in $\mathbb{Z}_p^*$).

Both the encryption parameters and the prime numbers are already computed at this point, and in this step a tool to select encryption parameters and voting option values, is implemented. This tool selects at random, a pair of encryption parameters and prime numbers from the sets generated in a pre-configuration phase, taking into account the number of values that will be needed to represent the voting options.

4.2.3 Generation of Control Components signing keys

Each Control Component generates a signing certificate per election to sign the information that they will generate during the configuration, the voting and the counting processes.

1. Call the RSA Key pair generation primitive and obtain the Control Component signing key pair: $(pk^s_{CCR_j}, sk^s_{CCR_j})$ / $(pk^s_{CCM_j}, sk^s_{CCM_j})$.

2. The Control Component is asked to introduce a KeyStore password.

3. The Control Component retrieves the Control Component CA private key $(CCRCA^j_{sk}/CCMCA^j_{sk})$ stored in the KeyStore.

4. Call the X509 certificate generation primitive with the Control Component CA private key $(CCRCA^j_{sk}/CCMCA^j_{sk})$ to generate the Control Component Signing X509 Certificate. The certificate should contain in the common name, the name of the component for which it is issued and the election event ID $(eeid)$.

5. Call the Random value generation primitive to generate a random password of length 26 chars in base 32.

6. The private RSA key is stored into a KeyStore and seal it with the password.

7. The KeyStore password must be stored encrypted using the Control Component encryption private key $(CCRe_{pk}^{j}/CCMe_{pk}^{j})$.

4.2.4 Generation of Authentication Context Information

In this step, the set of configuration information that should be stored in the Authentication Context is created. Some of this information, such as the Election Event ID $(eeid)$ and the CA certificates, have been already generated in previous steps.

First, the Authentication Token Signer key pair and certificate are generated:

1. Call the RSA Key pair generation primitive and obtain the Authentication Token Signer key pair (ATs_{pk}, ATs_{sk}).
2. Call the X509 certificate generation primitive to generate a certificate containing the Election Event ID $(eeid)$, the validity period, the target service identifier and the certificate name field values, signed with the Services CA private key (SCA_{sk}).
3. Call the Random value generation primitive to generate a random password of length 26 chars in base 32. This is the Authentication Token Signer Password.
4. Store the private RSA key into a KeyStore and seal it with the password.
5. Encrypt the Authentication Token Signer Password with the Authentication Context System public key.

Then, the following set-up parameters are established:

- Challenge-response expiration time
- Authentication Token expiration time
- Challenge length

Finally, the information is grouped as follows to distinguish the data to be sent to the Client Context and the data that will remain in the Authentication Context:

Authentication Voter Data
- Election Event ID $(\boldsymbol{eeid})$
- Election Event Root CA
- Services CA
- Authorities CA
- Credentials CA
- Authentication Token Signer certificate

Table 9- Authentication Voter Data

Authentication Context Data

- Election Event ID ($\boldsymbol{eeid}$)
- Authentication Token Signer KeyStore
- Authentication Token Signer Password (encrypted with the Authentication Context System public key)
- Setup parameters:
  - Challenge-response expiration time
  - Authentication Token expiration time
  - Challenge length

Table 10 - Authentication Context Data

4.2.5 Generation of Election Information Context Information

It generates the set of configuration information that should be stored in the Election Information Context. The set of information is the following:

Election Information Context Data

- Election Event ID ($\boldsymbol{eeid}$)
- Election Event Root CA
- Services CA
- Authorities CA
- Credentials CA
- Setup parameters:
  - Number of votes per Voting Card ID
  - Number of votes per Authentication Token ID

Table 11 - Election Information Context Data

4.2.6 Generation of Voting Workflow Context Information

Generates the information to be stored in the Voting Workflow Context:

Voting Workflow Context Data
- Number of confirmation attempts

Table 12 - Voting Workflow Context Data

4.3 Create Ballot

It assigns values to the voting options in a ballot. The following processes are executed.

- Assignation of voting option values.
- Assignation of attributes to voting options.

4.3.1 Assignation of voting option values

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

In this step, a value (prime number) is assigned to each of the voting options in the ballot. These values will be those encrypted at the voting phase.

The format of the complete ballot is detailed below.

4.3.1.1 Ballot format

The ballot contains the following fields:

Ballot

- Ballot ID ($\boldsymbol{bid}$)
- Default title
- Default description
- Alias
- Election Event ID ($\boldsymbol{eeid}$)
- Contest (*as many as the number of contests*)
    - Contest ID
    - Default title
    - Alias
    - Election Event ID ($\boldsymbol{eeid}$)
    - Template (*options or list and candidates*)
    - Full Blank *(true or false):* Setting this field to true, the voter can send the vote without selecting any option (election rules defined in “questions” do not apply).
    - Options: (As many as options in the contest)
        - ID
        - Representation: *prime number*
        - Attribute: Refers to one attribute of the next section and defines what kind of option it is (candidate, list, answer, write-in, blank candidate, blank list…).
    - Attributes
        - ID
        - Alias
        - Correctness*:* (True or false)
        - Related*:* Attributes related with this attribute (for instance, a specific answer is related with the attribute that represents its question).
    - Questions: (One per selectable option, that is, a list, a candidate, a question)
        - ID
        - Max: Maximum number of selections for this question
        - Min: Minimum number of selections for this question
        - Accumulation: Will be greater than 1 if the question can be selected more than once (for instance, vote twice for the same candidate).
        - writeIn: True if the question allows write-ins
        - blankAttribute: Attribute ID representing the blank attribute for this question.
        - writeInAttribute: Writein attribute ID representing the write in attribute for this question.
        - attribute: Attribute ID
        - fusions: Contains the alias of the attributes that represents the same.
    - encryptedCorrectnessRule: To be executed during the voting phase in the Election Information Context.
    - decryptedCorrectnessRule: To be executed over the decrypted vote.

- Status - Details - Synchronized - ballotBoxes - signedObject

Table 13 - Ballot

4.3.1.2 Write-ins

Depending on the type of the election, the ballot can allow the voter to enter k write-ins, that is, the voter can introduce some free-text. The protocol does not support individual verifiability of the content of the write-ins (it is impossible to generate Choice Return Codes from open text). However, the protocol can be used to prove the voter whether his/her intention has been casting a write-in instead of a selection of an explicit candidate or an explicit blank vote. This option only provides individual verifiability of explicit options when write-in is another option.

In this case, the ballot should contain one voting option for each write-in and a voting option value assigned to each one. This value does not represent the text inside the write-in but the fact that the voter has filled it in. The “write-in filled” voting options need to have the following labels/attributes:

- “write-in”
- An identifier of the write-in position to which they belong, so that a “write-in filled” voting option value can be related to a specific write-in field.

The text introduced by the voter should be encoded into a group element. Note that in order to be a group element, the value should be a quadratic residue.

4.3.2 Assignation of attributes to voting options

The voting options in the ballot are labelled with a set of attributes which allow the following:

- To print them with the correct format in the voter screen.
- To check that a vote is well-formed (hence, it contains a valid set of voting options). This check, also referred as vote correctness verification, is part of the cryptographic protocol. The vote correctness is checked at two levels.
    - The first level is when the vote is in plain text, and validations that need the full context of the election can be applied to it. These validations are done both before the vote is encrypted in the voting client and when the vote is decrypted.
    - The second level is done at the voting server over the encrypted vote. Vote correctness validations include validations about the write-ins, the existence of the return codes or the number of elements of the encrypted vote (among others).

Some of the attributes are used for verifying the vote correctness during the voting phase, which means that they are sent to the server, together with the vote. Such attributes are configured to contain a flag "correctness" set to true, while other attributes have this flag set to false.

The following attributes in the ballot have the flag "correctness = true":

- **Options**:
    - **Question ID attribute** (the attribute that refers to which question the voting option belongs to).
- **List and Candidates**:
    - **List attribute** (the attribute that says that this is a list).
    - **Candidate attribute** (the attribute that says that this is a candidate).

Other attributes such as (**blank, non_blank, writein**) have the flag "**correctness = false**".

4.4 Create Ballot Boxes

This process generates the configuration data specific to a set of Ballot Boxes. The Ballot Box configuration data defines how the votes are going to be encrypted (e.g., encryption parameters), stored (e.g., voting period) and verified (e.g., verification grace period).

Additionally, Ballot Boxes can be configured as a Test Ballot Box, which means that they can be downloaded at any time during the election process. One Ballot Box is identified by a unique Ballot Box Identifier (Ballot Box ID $(bbid)$) and is defined to contain one type of ballot, defined by the Ballot ID (bid). However, the same type of ballot may be stored in different Ballot Boxes.

For each of the Ballot Box identified by its corresponding Ballot Box ID $(bbid)$:

1) Call the RSA Key pair generation primitive and obtain the Ballot Box Signer key pair $(BBs_{pk}^{bb_{id}}, BBs_{sk}^{bb_{id}})$.
2) Call the X509 certificate generation primitive to generate a X.509 certificate containing the Ballot Box ID $(bbid)$, the validity period and the certificate name field values. This certificate is signed with the Services CA private key (SCA_{sk}).
3) Call the Random value generation primitive to generate a random password of length 26 chars in base 32.
4) Store the private RSA key into a KeyStore and seal it with the password.
5) Encrypt the password with the Election Information Context System public key.
6) Generate Ballot Box Information structure containing:

Ballot Box Information
- Ballot Box ID ($\boldsymbol{bbid}$) - Grace Period - Alias - Encryption Parameters - Election Key ID - Write-In Alphabet - Confirmation Required - Ballot Box Certificate - Ballot ID ($\boldsymbol{bid}$) - Test - Start date - End date - Election Event ID ($\boldsymbol{eeid}$)

Table 14 - Ballot Box Information

7) Generate a Ballot Box Context Data structure containing:

Ballot Box Context Data
- Ballot Box ID ($\boldsymbol{bbid}$) - Election Event ID ($\boldsymbol{eeid}$) - KeyStore - KeyStore password (encrypted with the Election Information Context System public key)

Table 15 - Ballot Box Context Data

8) Generate a Ballot Box Voter Data structure containing:

Ballot Box Voter Data
- Ballot ID ($\boldsymbol{bid}$) - Ballot Box ID ($\boldsymbol{bbid}$) - Election Event ID ($\boldsymbol{eeid}$) - Encryption parameters - Ballot Box certificate - Election key

Table 16 - Ballot Box Voter Data

Note that the information related with the Election key that is part of both the Ballot Box Information and Ballot Box Voter Data, cannot be included yet since it has not been generated yet. The Election key generation is explained in section 4.5.3.

4.5 Create Election key

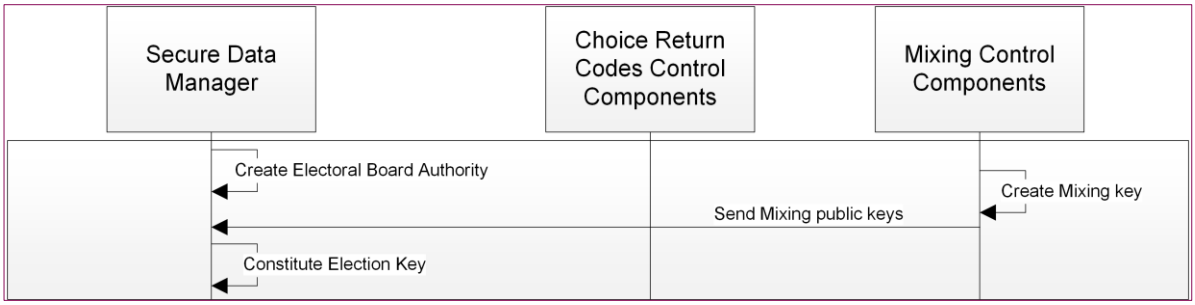

Figure 13 - Create Election key

The Election key is generated among the CCM_1, CCM_2, CCM_3 and the Electoral Board. The election public key (EL_{pk}) is set to be part of the information linked to one or more Ballot Boxes, and the votes which are intended to be stored in these Ballot Boxes will be encrypted with this key. The election private key (EL_{sk}) is kept by the Electoral Board members and the Control Components (CCM_1, CCM_2, CCM_3), which will use it to decrypt the votes collected in these Ballot Boxes after the voting phase ends. Without the decryption key, the votes cannot be decrypted, and the results cannot be obtained.

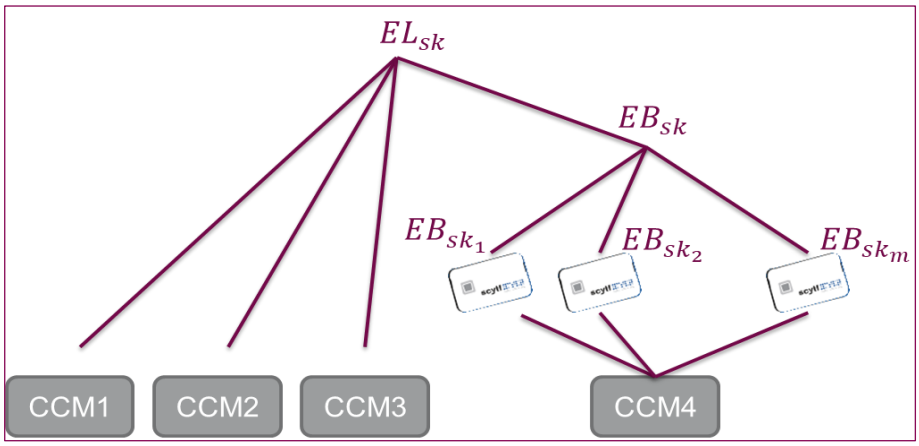

Figure 14 - Election key generation

4.5.1 Create Electoral Board Authority

The Electoral Board is an entity formed by several members who are responsible for decrypting the votes which have been collected in determined Ballot Boxes during an Election Event.

For this purpose, they are responsible for the generation of a pair of encryption/decryption keys.

The Electoral Board private key (EB_{sk}) key is split using a secret sharing scheme, and each piece is provided to one of the Electoral Board members. This prevents an Electoral Board member from being able to decrypt a set of votes on their own (for example, before the voting phase ends, to get some partial results information), without the agreement of a certain number of other members.

For each Electoral Board Authority, identified by the corresponding Electoral Authority ID, and given the threshold value:

1) Call the ElGamal Key pair generation primitive using the encryption parameters and obtain the Electoral Board key pair: $(EB_{pk}, EB_{sk}) = (g^{EB_{SK}}, EB_{sk})$

2) Call the Shamir Threshold Secret Sharing split algorithm to split the Electoral Board private key (EB_{sk}) into as many pieces as indicated by the number of members of the EB, and with the threshold defined.

3) Call the Digital signature generation primitive to sign every share with the Authorities CA private key (ACA_{sk})

4) Generate the Electoral Authority structure containing:

Electoral Authority Data
- Electoral Authority ID - Electoral authority public key ($\boldsymbol{EB_{pk}}$)

Table 17 - Electoral Authority Data

As mentioned before, the Electoral Board Authority is generated by the *Election Keys Generation* module of the *Print Office* component using the SDM offline software.

4.5.2 Create Control Components Mixing key

In this step the Control Components contributions to the Election key pair (EL_{pk}, EL_{sk}) are generated.

For each Electoral Authority and each Control Component CCM_j where $j \in \{1,2,3\}$:

1. Call the ElGamal Key pair generation primitive using the encryption parameters provided and obtain the CCM_j Mixing key pair: (g^{x_j}, x_j).

2. Call the Digital signature generation primitive to sign the CCM_j Mixing public key (g^{x_j}) together with the Electoral Authority ID and the Election Event ID $(eeid)$ using the CCM_j signing private key $(sk^s_{CCM_j})$.

3. Encrypt CCM_j Mixing private key (x_j) using the CCM_j encryption public key $(CCMe^j_{pk})$.

The CCM_j mixing public key (g^{x_j}) and its signature are sent to the Secure Data Manager (Print Office) to constitute the Election public key (EL_{pk}) once the Electoral Board has been created.

4.5.3 Constitute Election key

Once the Control Components and the Electoral Board have generated their own keys, the Election key can be constituted following the next steps:

1. Validate the Control Components certificate chains:
    - [CCM_1 signing certificate, CCM_1 CA Certificate, Platform Root CA]
    - [CCM_2 signing certificate, CCM_2 CA Certificate, Platform Root CA]

- [CCM_3 signing certificate, CCM_3 CA Certificate, Platform Root CA]

2. Validate the following signatures using the corresponding CCM_j signing certificate:
   - CCM_1 mixing public key (g^{x_1}) signature
   - CCM_2 mixing public key (g^{x_2}) signature
   - CCM_3 mixing public key (g^{x_3}) signature
3. Generate the Election public key:

$$EL_{pk} = EB_{pk} \cdot \prod_{j=1}^{3} g^{x_j} = g^{EB_{sk}+\sum_{j=1}^{3} x_j}$$

4. Generate the Election Key structure containing:

Election Key Data
- Election Key ID - Election public key ($\boldsymbol{EL_{pk}}$)

Table 18 - Election Key Data

5. Fill the Election Key ID field in each of the Ballot Box Information items (see Table 14 - Ballot Box Information).
6. Fill the Election Key ID field in each of the Ballot Box Voter Data items

4.5.3.1 Election key when write-ins are allowed

In case the election type allows the voter to enter free text (write-in), additional ciphertexts resulting in the encryption of these text values are added to the vote message. The key to encrypt them should be different from the one used to encrypt the votes to prevent attacks.

For this purpose, the Electoral Board Key and the Control Components Mixing Key should have at least as many components as the maximum number of write-ins (k) to be filled in a ballot managed by such Electoral Authority, plus one.

$$EL_{pk} = (EL_{pk}^{(1)}, \dots, EL_{pk}^{(m)})$$

4.6 Protocol Setup algorithm

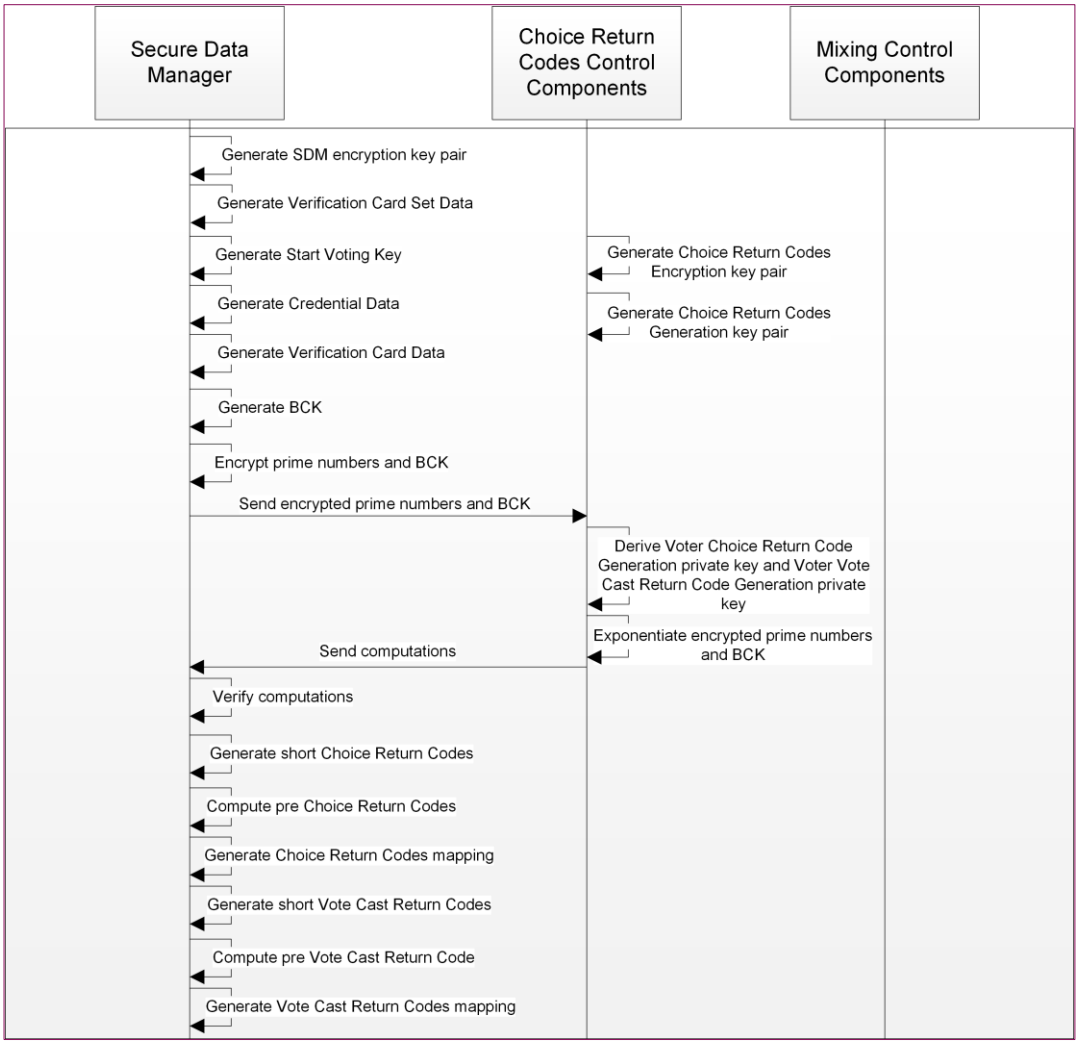

Figure 15 - Protocol Setup algorithm

During the execution of the Setup algorithm the following information is generated:

- SDM encryption key pair
- Verification Card Set Data
- Voting Card Set Data
    - Start Voting Key (SVK_{id}), Credential ID (c_{id}) and KeyStore symmetric encryption key ($KSkey_{id}$)
    - Credential Data
    - Verification Card Data
    - Verification Card Codes

While the SDM encryption key pair and the Verification Card Set Data are generated entirely in the Secure Data Manager (Print Office), the Voting Card Set Data is generated among the Secure Data Manager (Print Office) and the Control Components.

The following tables contain a summary of the keys and identifiers generated by either the Secure Data Manger (Print Office) or the Control Components, during the execution of the Setup algorithm.

Secure Data Manager (Print Office)
Generates per Election Event: - Secure Data Manager encryption key pair ($\boldsymbol{pk_{SDM}, sk_{SDM}}$)
Generates per Verification Card Set: - Verification Card Set ID ($\boldsymbol{vcs_{id}}$) - Codes Secret Key ($\boldsymbol{C_{sk}}$) - Vote Cast Return Code Signer key pair ($\boldsymbol{VCCs_{pk}, VCCs_{sk}}$) - Choice Return Codes Encryption Public key ($\boldsymbol{pk_{CCR}}$) *(this key is computed from the* $\boldsymbol{CCR_j}$ *Choice Return Codes Encryption public keys)* - Verification Card Set Issuer key pair ($\boldsymbol{VCI_{pk}, VCI_{sk}}$)
Generates per Voter: - Voting Card ID ($\boldsymbol{vcd_{id}}$) - Verification Card ID ($\boldsymbol{vc_{id}}$) - Start Voting Key ($\boldsymbol{SVK_{id}}$) - Ballot Casting Key ($\boldsymbol{BCK^{id}}$) - Short Choice Return Codes ($\boldsymbol{CC_1^{id}, \dots, CC_n^{id}}$) - Short Vote Cast Return Code ($\boldsymbol{VCC^{id}}$) - Credential ID Authentication key pair ($\boldsymbol{K^a_{C_{id}}, k^a_{C_{id}}}$) - Credential ID Signing key pair ($\boldsymbol{K^s_{C_{id}}, k^s_{C_{id}}}$) - Verification Card key pair ($\boldsymbol{K_{id}, k_{id}}$)
Computes per Voter: - pre-Choice Return Codes ($\boldsymbol{pC_1^{id}, \dots, pC_n^{id}}$) - long Choice Return Codes ($\boldsymbol{lCC_1^{id}, \dots, lCC_n^{id}}$) - Choice Return Code encryption symmetric key ($\boldsymbol{skcc_1^{id}, \dots, skcc_n^{id}}$) - pre-Vote Cast Return Code ($\boldsymbol{pVCC^{id}}$) - long Vote Cast Return Code ($\boldsymbol{lVCC^{id}}$) - Vote Cast Return Code encryption symmetric key ($\boldsymbol{skvcc^{id}}$)
Derives per Voter: - Credential ID ($\boldsymbol{c_{id}}$) - KeyStore symmetric encryption key ($\boldsymbol{KSkey_{id}}$)

Table 19 - Keys and identifiers generated by the SDM during the Setup algorithm

Control Component CCR_j
Generates per Verification Card Set: - $\boldsymbol{CCR_j}$ Choice Return Codes Encryption key pair $(\boldsymbol{pk_{CCR_j}}, \boldsymbol{sk_{CCR_j}})$ - $\boldsymbol{CCR_j}$ Choice Return Codes Generation key pair $(\boldsymbol{g^{k'_j}}, \boldsymbol{k'_j})$
Derives per Voter: - Voter Choice Return Codes generation key pair $(\boldsymbol{K^j_{id}}, \boldsymbol{k^j_{id}})$ - Vote Cast Return Code generation key pair $(\boldsymbol{Kc^j_{id}}, \boldsymbol{kc^j_{id}})$

Table 20 - Keys and identifiers generated by the CCR_j during the Setup algorithm

4.6.1 Generate SDM encryption key pair

The Secure Data Manager (Print Office) calls the ElGamal Key pair generation primitive and obtains the SDM encryption key pair (pk_{SDM}, sk_{SDM}) that will be used to encrypt sensitive information that is sent between the Control Components and the SDM (Print Office).

4.6.2 Generate Verification Card Set Data

For each Ballot Box created, a set of voting cards is generated. This set of voting cards defines the resources that will be assigned to the voter and used by them to cast their vote. Voters who use voting cards from a specific set will cast votes into a specific Ballot Box. Correspondingly, a Verification Card Set is generated, which is intended to group the information corresponding to the cast-as-intended verification process for the voters in the same Voting Card and Verification Card sets, given that they have one-to-one correspondence.

For each Verification Card Set:

1) Generate a Verification Card Set Identifier (Verification Card Set ID).
2) Generate the Codes Secret Key (C_{sk}):
    - Call the Symmetric key generation primitive to generate a random secret key and set it to be the Codes Secret Key (C_{sk})
    - Call the Random value generation primitive to generate a random password of length 26 chars in base 32.
    - Store the Codes Secret Key KeyStore and seal it with the password generated.
3) Generate the Vote Cast Return Code Signer keypair and certificate:
    - Call the RSA Key pair generation primitive and obtain the Vote Cast Return Code Signer key pair $(VCCs_{pk}, VCCs_{sk})$
    - Call the X509 certificate generation primitive to generate a X.509 certificate containing the Election Event ID $(eeid)$, the Verification Card Set ID (vcs_{id}), the validity period and the certificate name field values

4.6.3 Create Voting Card Set

For each Ballot Box created, a set of voting cards is generated. This set of voting cards defines the resources that will be assigned to the voter and used by them to cast their vote. Voters who use voting cards from a specific set will cast votes into a specific Ballot Box.

- First, a unique Voting Card Set Identifier ($vcds_{id}$) is generated.

Then, the following processes are executed:

- Generation of Start Voting Key (SVK_{id}), Credential ID (c_{id}) and KeyStore symmetric encryption key ($KSkey_{id}$)KeyStore.
- Generation of Credential Data.
- Generation of Verification Card Data.
- Verification Card Codes generation.

4.6.3.1 Generation of Start Voting Key, Credential ID and KeyStore Password

The Start Voting Key (SVK_{id}) is the password entered by the voter in the Voter Portal. This password will be used to authenticate the voter and to give him permission to cast a vote.

From this Start Voting Key (SVK_{id}), two values are derived; the Credential ID (c_{id}), which identifies a set of resources (i.e., voting credentials) to be assigned to the holder of that start voting key, and the KeyStore symmetric encryption key ($KSkey_{id}$), which will be used to open the corresponding KeyStores.

For each Voting Card:

1. Generate a Voting Card ID (vcd_{id}).
2. Call the Random value generation primitive to generate the Start Voting Key (SVK_{id}) as a random value of length 20 chars in base 32.
3. Call the Password-based key derivation function with the following inputs:
    - Password: Start Voting Key (SVK_{id}).
    - Salt: concatenation of the string ‘credentialid’ and the Election Event ID ($eeid$).

    The result is the Credential ID (c_{id}).
4. Call the Password-based key derivation function with the following inputs:
    - Password: Start Voting Key (SVK_{id}).
    - Salt: concatenation of the string ‘KeyStorepin’ and the Election Event ID ($eeid$).

    The result is the KeyStore symmetric encryption key ($KSkey_{id}$).

After all the Voting Card have been generated, the following list of pairs are created:

- $(c_{id} - KSkey_{id})$

- $(c_{id} - SVK_{id})$

4.6.3.1.1 Extended authentication

For the voter to be authenticated, it is also possible to require additional authentication values and a challenge question (for instance, the year of birth). The voter will introduce the Start Voting Key (SVK_{id}) in the Voter Portal and this extra information.

In this situation, additional information will be generated besides the Start Voting key (SVK_{id}), the Credential ID (c_{id}) and the KeyStore password $(KSkey_{id})$. In order to generate this additional information, the following data should be provided:

- List of pairs (VoterAlias – Challenge answer)
- Number of additional authentication values and its length.
- List of pairs $(c_{id} - SVK_{id})$, computed in the previous step.

Then, for each Voting Card:

1) Relate the Voting Card and the Start Voting Key (SVK_{id}) to a pair of (VoterAlias – Challenge answer).
2) Generate an Authentication Key from the additional authentication values. In case no additional information is required to be authenticated, call the Random value generation primitive to generate a random password of length 20 chars in base 32. The result will be the Authentication Key.
3) Call the Password-based key derivation function with the following inputs:
    - Password: Authentication Key value.
    - Salt: concatenation of the string ‘authid’ and the Election Event ID $(eeid)$.

    The result is the Authentication ID.
4) Call the Password-based key derivation function with the following inputs:
    - Password: Authentication Key value.
    - Salt: concatenation of the string ‘authpassword’ and the Election Event ID $(eeid)$.

    The result is the password.
5) Call the Symmetric encryption primitive to encrypt the Start Voting Key (SVK_{id}) using the derived password.
6) Call the Password-based key derivation function with the following inputs:
    - Password: Challenge Answer value.
    - Salt: call the Random value generation primitive to compute a random salt of 256 bits.

After the extended authentication information for all the generated Voting Cards, the following list of pairs are created:

- (Credential ID (c_{id}) – Authentication ID).
- (VoterAlias – Authentication ID).
- (Authentication ID – Authentication Key).
- (Authentication ID – PBKDF2(ChallengeAnswer, Salt) – Salt – Encrypted (SVK_{id})).

4.6.3.2 Generation of Credential Data

Generation of credential data, which is needed to authenticate into the platform and cast a vote. A Credential Data item is associated to one Voting Card.

For each Credential ID (c_{id}):

1) Call twice the RSA Key pair generation primitive and obtain the:
   - Credential ID Authentication key pair ($K^a_{C_{id}}, k^a_{C_{id}}$)
   - Credential ID Signing key pair ($K^s_{C_{id}}, k^s_{C_{id}}$)
2) Call twice the X509 certificate generationprimitive to generate two X.509 certificates, one for each RSA public key, containing the Election Event ID ($eeid$), the Credential ID (c_{id}), the validity period and the certificate name field values. One will be intended for authentication, and the other for digital signatures. Therefore, include 'Auth' or 'Sign' respectively in the certificate field.
3) Digitally sign the certificates using the Credentials CA private key (CCA_{sk}).
4) Store the private RSA keys and the certificates into a KeyStore and seal it with the corresponding KeyStore password ($KSkey_{id}$).

Generate the Credential Data structure with the following information:

Credential Data

- List of pairs (Credential ID ($\boldsymbol{c_{id}}$) - Credential KeyStore), where each KeyStore contains:
  - Certificate for the authentication public key
  - Certificate for the signing public key
  - Credential ID Authentication private key ($\boldsymbol{k^a_{C_{id}}}$)
  - Credential ID Signing private key ($\boldsymbol{k^s_{C_{id}}}$)

Table 21 - Credential Data

4.6.3.3 Generation of Verification Card Data

Generation of the data that will be needed to compute the *Encrypted vote* and the *Confirmation message* at the voting client context during the voting phase. This information is generated both in the Control Components and in the Secure Data Manager (Print Office).

For each Verification Card Set ID (vcs_{id}), each Control Component CCR_j:

1) Generates the CCR_j Choice Return Codes Encryption key pair:
   - Calls the ElGamal Key pair generation primitive and generates a key pair with ψ components, where ψ is the maximum number of options a voter can select:

     $$pk_{CCR_j}: \left\{\left(pk_{CCR_j}^{(1)}, sk_{CCR_j}^{(1)}\right), \ldots, \left(pk_{CCR_j}^{(\psi)}, sk_{CCR_j}^{(\psi)}\right)\right\}$$

     where $pk_{CCR_j}^{(i)} = g^{sk_{CCR_j}^{(i)}}$.
   - The public part of every key pair is signed together with the Verification Card Set ID (vcs_{id}) and the Election Event ID ($eeid$), using the CCR_j signing private key ($sk_{CCR_j}^{s}$).
   - Encrypts the private key using the CCR_j encryption public key.
2) Generates the CCR_j Choice Return Codes Generation key pair:
   - Calls the ElGamal Key pair generation primitive and generates their own key pair: $(g^{k'_j}, k'_j)$.
   - Encrypts the private key using the CCR_j encryption public key (generated in section 3.5.3).
3) Sends the CCR_j Choice Return Codes Encryption public key: $\left(pk_{CCR_j}^{(1)}, \ldots, pk_{CCR_j}^{(\psi)}\right)$ and its signature to the Secure Data Manager (Print Office).

For each Verification Card Set ID (vcs_{id}), the Secure Data Manager (Print Office):

1) Receives the following information from the Control Components:
   - CCR_1 Choice Return Codes Encryption public key: $\left(pk_{CCR_1}^{(1)}, \ldots, pk_{CCR_1}^{(\psi)}\right)$ and its signature.
   - CCR_2 Choice Return Codes Encryption public key: $\left(pk_{CCR_2}^{(1)}, \ldots, pk_{CCR_2}^{(\psi)}\right)$ and its signature.
   - CCR_3 Choice Return Codes Encryption public key: $\left(pk_{CCR_3}^{(1)}, \ldots, pk_{CCR_3}^{(\psi)}\right)$ and its signature.
   - CCR_4 Choice Return Codes Encryption public key: $\left(pk_{CCR_4}^{(1)}, \ldots, pk_{CCR_4}^{(\psi)}\right)$ and its signature.
2) Generates a Verification Card Set Issuer keypair:
   - Call the RSA Key pair generation primitive and obtain the Verification Card Set Issuer key pair (VCI_{pk}, VCI_{sk}).

- Call the X509 certificate generation primitive to generate a X.509 certificate for the RSA public key, containing the Election Event ID $(eeid)$, the Verification Card Set ID (vcs_{id}), the validity period and the certificate name field values.

3) Verifies the signatures of the CCR_j Choice Return Codes Encryption public keys $(pk_{CCR_1}, \dots, pk_{CCR_4})$ and multiplies them to obtain the Choice Return Codes Encryption public key (pk_{CCR}):

$$\begin{cases} pk_{CCR}^{(1)} = pk_{CCR_1}^{(1)} \cdot \dots \cdot pk_{CCR_4}^{(1)} = g^{sk_{CCR_1}^{(1)} + \dots + sk_{CCR_4}^{(1)}} \\ \vdots \\ pk_{CCR}^{(\psi)} = pk_{CCR_1}^{(\psi)} \cdot \dots \cdot pk_{CCR_4}^{(\psi)} = g^{sk_{CCR_1}^{(\psi)} + \dots + sk_{CCR_4}^{(\psi)}} \end{cases}$$

The corresponding private key will be $sk_{CCR} = (sk_{CCR}^{(1)}, \dots, sk_{CCR}^{(\psi)})$.

4) Stores the Choice Return Codes Encryption public key (pk_{CCR}), the individual CCs Choice Return Codes public keys $(pk_{CCR_1}, \dots, pk_{CCR_4})$ and their signatures. This information will be uploaded later to the voting channel.

5) For each Credential ID (c_{id}) in the Verification Card Set, generate a Verification Card Data item in the following way:

- Generates a random identifier Verification Card ID (vc_{id}).
- Calls the ElGamal Key pair generation primitive using the encryption parameters provided and obtain the Verification Card key pair: (K_{id}, k_{id}).
- Calls the Digital signature generation primitive to sign the Verification Card public key (K_{id}), the Election Event ID $(eeid)$ and the Verification Card ID (vc_{id}) using the Verification Card Set Issuer private key (VCI_{sk}).
- Stores the Verification Card private key (k_{id}) into a KeyStore $(VCks_{id})$ and seal it with the corresponding password $(KSkey_{id})$.

6) Generates the Verification Card Data set:

Verification Card Data
Contains one row per Verification Card ID with the following information: - Verification Card ID $(\boldsymbol{vc_{id}})$ - Verification Card Set ID $(\boldsymbol{vcs_{id}})$ - Election Event ID $(\boldsymbol{eeid})$ - Verification Card KeyStore $(\boldsymbol{VCks_{id}})$, containing the Verification Card private key $(\boldsymbol{k_{id}})$ - Signed Verification Card public key $(\boldsymbol{K_{id}})$

Table 22 - Verification Card Data

7) Once all the Verification Card Data items have been generated, creates the following list of pairs

- (Verification Card private key (k_{id}) - Verification Card ID (vc_{id})).
- (Verification Card ID (vc_{id}) – Voting Card ID (vcd_{id})).

4.6.3.4 Verification Card Codes generation

In this step, the codes to be printed in the verification card for the voter are generated. These codes are the following:

- Choice Return Codes ($CC_1^{id}, \dots, CC_n^{id}$), which are linked to a ballot's set of voting options ($v_1, \dots, v_n$),
- Ballot Casting Key (BCK^{id}), and
- Vote Cast Return Code (VCC^{id}) which are used to confirm and check the confirmation of the vote casting.

Additionally, a Codes Mapping Table is created to provide the link between the voting options ($v_1, \dots, v_n$) and the Choice Return Codes ($CC_1^{id}, \dots, CC_n^{id}$), and between the Ballot Casting Key (BCK^{id}) and the Vote Cast Return Code (VCC^{id}).

The Verification Card Codes and the corresponding Mapping Table are generated for a specific ballot. The process below is performed for all the Verification Card IDs (vc_{id}) in a specific Verification Card Set (vcs_{id}) and is an interaction between the Secure Data Manager (Print Office) and the Control Components CCR_j.

For each Verification Card Set (vcs_{id}), the SDM (Print Office):

1) Calls the ElGamal encryption primitive as many times as voting options in the ballot, and using the Secure Data Manager public key (pk_{SDM}), the output is:

$$\{E_{pk_{SDM}}(v_1), E_{pk_{SDM}}(v_2), \dots, E_{pk_{SDM}}(v_n)\}$$

2) For each Verification Card ID (vc_{id}) in the Verification Card Set:
   - Calls the Random value generation primitive to generate a random 8-digit value and computes from it a one-digit checksum using the EAN13 standard. Let the concatenation of the two values be the Ballot Casting Key (BCK^{id}).
   - Squares the BCK^{id} and computes the exponentiation to the Verification Card private key k_{id}:

$$CM^{id} = (BCK^{id})^{2k_{id}}$$

   - Compute a hash of CM^{id} and encrypts the squared result calling the ElGamal encryption primitive with the SDM encryption public key pk_{SDM}:

$$E_{pk_{SDM}}(Hash(CM^{id})^2).$$

3) Calls the Digital signature generation primitive to sign the encrypted prime numbers and the list of encrypted $Hash(CM^{id})^2$ using the Administration Board private key (AB_{sk}) and sends the signed information to the Control Components.

For each the Verification Card Set, each Control Component:

1) Receives from the SDM (Print Office) the following information:
   - Encrypted prime numbers: $\{E_{pk_{SDM}}(v_1), E_{pk_{SDM}}(v_2), \dots, E_{pk_{SDM}}(v_n)\}$
   - List of Encrypted hashes: $E_{pk_{SDM}}(Hash(CM^{id})^2)$ corresponding to each Verification card ID (vc_{id}) in the Verification Card Set.
   - Signature of encrypted prime numbers and encrypted $Hash(CM^{id})^2$.
   - Administration Board X509 Certificate and Tenant CA X509 Certificate.
   - List of Verification Card IDs (vc_{id}) included in the Verification Card Set.
2) For each Verification Card ID (vc_{id}), computes the following values:
   - The derivation of the Voter Choice Return Code generation private key (k_{id}^j) from the CCR_j Choice Return Codes generation private key (k'_j) and the Verification Card ID (vc_{id}):
     - Compute $k_{id}^j = KDF(vc_{id}||k'_j,\ p\ length)$ calling the Key Derivation Function: KDF1 specification primitive.
     - Truncate the result to have 2047 bits.
     - Check that $1 \leq k_{id}^j \leq q - 1$
     - If the derived value is equal or greater than q, compute again a derivation but using as input the derived value $KDF(k_{id}^j,\ p\ length)$.
     - k_{id}^j is the Voter Choice Return Code generation private key.
     - Compute the Voter Choice Return Code generation public key (K_{id}^j) associated to the Voter Choice Return Code generation private key (k_{id}^j) as the exponentiation of the generator g to k_{id}^j: $K_{id}^j = g^{k_{id}^j}$.

     Notice that as k_{id}^j is computed using the Verification Card ID(vc_{id}), it is specific per voter.
   - The exponentiation of $E_{pk_{SDM}}(v_1)$, $E_{pk_{SDM}}(v_2), \dots, E_{pk_{SDM}}(v_n)$ to the corresponding Voter Choice Return Code generation private key (k_{id}^j):

$$\left\{E_{pk_{SDM}}(v_1)^{k_{id}^j}, E_{pk_{SDM}}(v_2)^{k_{id}^j}, \dots, E_{pk_{SDM}}(v_n)^{k_{id}^j}\right\}$$

- Call the Exponentiation proof generation primitive to compute a proof of knowledge of the exponent k_j^{id}. The inputs of the primitive are the following:
  - Base elements (group elements): $[g, E_{pk_{SDM}}(v_1),\ E_{pk_{SDM}}(v_2), \dots, E_{pk_{SDM}}(v_n)]$
  - Exponents: $[k_{id}^j]$
  - Public input (group elements):

    $$[K_{id}^j, E_{pk_{SDM}}(v_1)^{k_{id}^j}, E_{pk_{SDM}}(v_2)^{k_{id}^j}, \dots, E_{pk_{SDM}}(v_n)^{k_{id}^j}]$$

  - Additional information "$ExponentiationProof$"
  - Mathematical group (p, q, g)

  The proof is logged to be validated during the verification phase.

- The derivation of the Voter Vote Cast Return Code generation private key (kc_{id}^j) from the CCR_j Choice Return Codes generation private key (k_j') and the Verification Card ID, the confirm text padding and the Verification Card ID (vc_{id}):
  - Compute $kc_{id}^j = KDF\big(vc_{id}||\text{confirm}||k_j',\ p\ length\big)$ calling the Key Derivation Function: KDF1 specification primitive.
  - Truncate the result to have 2047 bits.
  - Check that $1 \leq kc_{id}^j \leq q - 1$
  - If the derived value is equal or greater than q, compute again a derivation but using as input the derived value $KDF\big(kc_{id}^j,\ p\ length\big)$.
  - kc_{id}^j is the Voter Vote Cast Return Code generation private key.
  - Compute the Voter Vote Cast Return Code generation public key associated to the Voter Vote Cast Return Code generation private key (kc_{id}^j) as the exponentiation of the generator g to kc_{id}^j: $Kc_{id}^j = g^{kc_{id}^j}$.

  Notice that as kc_{id}^j is computed using the Verification Card ID (vc_{id}), it is specific per voter.

- The exponentiation of $E_{pk_{SDM}}(Hash(CM^{id})^2)$ to the corresponding kc_{id}^j.

  $$E_{pk_{SDM}}(Hash(CM^{id})^2)^{kc_{id}^j}$$

- Call the Exponentiation proof generation primitive to compute a proof of knowledge of the exponent kc_{id}^j. The inputs of the primitive are the following:
  - Base elements (group elements): $[g, E_{pk_{SDM}}(Hash(CM^{id})^2)]$

- Exponents: $[kc_{id}^{j}]$
- Public input (group elements): $[Kc_{id}^{j}, E_{pk_{SDM}}(Hash(CM^{id})^2)^{kc_{id}^{j}}]$
- Additional information "$ExponentiationProof$"
- Mathematical group (p, q, g)

The proof is logged to be validated during the verification phase.

- Calls the Digital signature generation primitive to sign the result of the exponentiations and the Voter Choice Return Code generation public key (Kc_{id}^{j}), using the CCR_j signing private key $(sk_{CCR_j}^{s})$.

3) Sends the computations signed to the SDM (Print Office)

For each Verification Card Set, the SDM (Print Office):

1) Receives from each Control Component the following information (notice that every element of the list mentioned below corresponds to one Verification Card ID (vc_{id}) of the Verification Card Set):
   - List of exponentiated encrypted primer numbers:

$$\left\{E_{pk_{SDM}}(v_1)^{k_{id}^{j}}, E_{pk_{SDM}}(v_2)^{k_{id}^{j}}, \dots, E_{pk_{SDM}}(v_n)^{k_{id}^{j}}\right\}$$

   - List of exponentiated encrypted squared hashes of Confirmation Messages: $E_{pk_{SDM}}(Hash(CM^{id})^2)^{k_{id}^{j}}$
   - List of Voter Choice Return Code generation public key: K_{id}^{j}
   - List of Voter Vote Cast Return Code generation public key: Kc_{id}^{j}
   - Signature of the information detailed above.

   And does the following actions:

   - Verify the signature of the received information using the CCR_j signing certificate. Store the public keys to be uploaded later to the Vote Verification Context. The signatures of these keys are also kept by the SDM (which is considered as part of the Bulletin Board) to be used by the Verifier during an audit process.
   - For each Verification Card ID (vc_{id}) in the Verification Card Set:
     - Call the Random value generation primitive to compute a 4-digit random (short) Choice Return Code (CC_i^{id}) for each voting option in the ballot.
     - Multiply the values received from the control components:

$$\{\prod_{j=1}^{4} E_{pk_{SDM}}(v_1)^{k_{id}^j}, \dots, \prod_{j=1}^{4} E_{pk_{SDM}}(v_n)^{k_{id}^j}\}$$

- Compute the exponentiation of each element above to the Verification card private key (k_{id}):

$$\{E_{pk_{SDM}}(v_1)^{k_{id}\cdot\Sigma_{j=1}^{4} k_{id}^j}, \dots, E_{pk_{SDM}}(v_n)^{k_{id}\cdot\Sigma_{j=1}^{4} k_{id}^j}\}$$

- For each one of the ciphertexts computed in the previous steps, call the ElGamal decryption primitive with input the ciphertext and the Secure Data Manager private key (sk_{SDM}). Finally, obtain the pre-Choice Return Codes $(pCC_1^{id}, \dots, pCC_n^{id}) = (v_1^k, \dots, v_n^k)$ where v_i^k is:

$$pCC_i^{id} = v_i^k = v_i^{k_{id}\cdot\Sigma_{j=1}^{4} k_{id}^j}$$

- For each pre-Choice Return Code pCC_i^{id}, concatenate it with the Verification Card ID (vc_{id}), the Election Event ID $(eeid)$ and the corresponding voting option attributes with the flag "correctness = true". Call the Hash generation primitive with input the concatenated value. The result is the long Choice Return Code:

$$lCC_i^{id} = Hash(pCC_i^{id} || vc_{id} || eeid || \{attributes\})$$

- For each long Choice Return Code (lCC_i^{id}), concatenate it with the Codes Secret Key (C_{sk}), compute a hash of the result and call the Key Derivation Function: KDF1 specification primitive to generate Choice Return Code encryption symmetric key :

$$skcc_i^{id} = KDF\big(Hash(lCC_i^{id} || C_{sk}), 256\ bits\big)$$

- Call the Symmetric encryption primitive to encrypt the corresponding random Choice Return Code (CC_i^{id}) with the symmetric key $skcc_i^{id}$ generated in the step before: $E(CC_i^{id}, skcc_i^{id})$.

- Call the Hash generation primitive for each long choice return lCC_i^{id} and create an entry in the table containing the encrypted Choice Return Code, where the entry key is the Hash of the long return code lCC_i^{id} : $Hash\big(lCC_i^{id}\big) – Enc\big(CC_i^{id}, skcc_i^{id}\big)$.

- Call the Random value generation primitive to compute an 8-digit random Vote Cast Return Code (VCC^{id}).

- Call the Digital signature generation primitive to sign the Vote Cast Return Code (VCC^{id}) and the Verification Card ID (vc_{id}) using the Vote Cast Return Code Signer private key $(VCCs_{sk})$.

$$s_{VCC_{id}} = Sign(VCC, VCCs_{sk})$$

- Multiply the values received from the Control Components and obtain the encrypted pre-Vote Cast Return Code:

$$\prod_{j=1}^{4} E_{pk_{SDM}}(Hash(CM^{id})^2)^{kc_{id}^j} = E_{pk_{SDM}}(Hash((BCK^{id})^{2k_{id}})^2)^{\sum_{j=1}^{4} kc_{id}^j}$$

$$= E_{pk_{SDM}}(pVCC^{id})$$

- Call the ElGamal decryption primitive to decrypt the pre-Vote Cast Return Code: $D_{sk_{SDM}}(E_{pk_{SDM}}(pVCC^{id}))$.
- Call the Hash generation primitive with input the concatenation of the pre-Vote Cast Return Code with the Verification Card ID (vc_{id}) and the Election Event ID ($eeid$). The result is the long Vote Cast Return Code:

$$lVCC^{id} = Hash(pVCC^{id} || VC_{id} || EEID)$$

- Concatenate the long Vote Cast Return Code ($lVCC^{id}$) with the Codes Secret Key (C_{sk}), and compute a hash of the result. Call the Key Derivation Function: KDF1 specification primitive to generate Vote Cast Return Code encryption symmetric key : $skvcc^{id} = KDF(Hash(lVCC^{id} || C_{sk}),\ 256\ bits)$.
- Call the Symmetric encryption primitive to encrypt the VCC^{id} and the signed VCC^{id} ($s_{VCC_{id}}$) using the symmetric key: $E(VCC^{id} || s_{VCC_{id}}, skvcc^{id})$.
- Call the Hash generation primitive with input the $lVCC^{id}$ and create an entry in the table containing the encrypted VCC^{id} and signed VCC^{id}, where the entry key is the Hash of the long Vote Cast Return Code $lVCC^{id}$: $Hash(lVCC^{id}) - Enc\left((VCC^{id} || s_{VCC_{id}}), skvcc^{id}\right)$.

Once all the computations are finished, the Codes Mapping Table contains entries for each of the Verification Card IDs in the Verification Card Set. These entries correspond to the Choice Return Codes and the Vote Cast Return Code assigned to each Verification Card ID (vc_{id}).

Additionally, the following data structure is created for each one of the Verification Card IDs in the Verification Card Set.

Verification Card Codes
- Verification Card ID ($\boldsymbol{vc_{id}}$) - Election Event ID ($\boldsymbol{eeid}$) - List of [Choice Return Codes ($\boldsymbol{CC_1^{id}}, \dots, \boldsymbol{CC_n^{id}}$) – ballot voting option identifiers ($\boldsymbol{v_1}, \dots, \boldsymbol{v_n}$)] - Ballot Casting Key ($\boldsymbol{BCK^{id}}$). - Vote Cast Return Code ($\boldsymbol{VCC^{id}}$)

Table 23 - Verification Card Codes

4.6.4 Verify Setup

During Setup the auditors must verify, for each Verification Card ID (vc_{id}), that the exponentiation proofs computed by the Control Components during the exponentiation of the encrypted prime numbers and the encrypted hashes of the confirmation messages, are correct. The following information should be retrieved in order to perform the validations:

- Exponentiation proofs stored in the Control Components Secure Loggers.
- The encrypted prime numbers $\{E_{pk_{SDM}}(v_1),\ E_{pk_{SDM}}(v_2), \dots, E_{pk_{SDM}}(v_n)\}$.
- For each Verification Card ID (vc_{id}):
    - $E_{pk_{SDM}}(Hash(CM^{id})^2)$
    - $E_{pk_{SDM}}(Hash(CM^{id})^2)^{kc_{id}^j}$
    - Exponentiated encrypted primer numbers $\{E_{pk_{SDM}}(v_1)^{k_{id}^j}, E_{pk_{SDM}}(v_2)^{k_{id}^j}, \dots, E_{pk_{SDM}}(v_n)^{k_{id}^j}\}$
    - CCR_1 Voter Choice Return Code generation public key (K_{id}^1)
    - CCR_2 Voter Choice Return Code generation public key (K_{id}^2)
    - CCR_3 Voter Choice Return Code generation public key (K_{id}^3)
    - CCR_4 Voter Choice Return Code generation public key (K_{id}^4)
    - CCR_1 Voter Vote Cast Return Code generation public key (Kc_{id}^1)
    - CCR_2 Voter Vote Cast Return Code generation public key (Kc_{id}^2)
    - CCR_3 Voter Vote Cast Return Code generation public key (Kc_{id}^3)
    - CCR_4 Voter Vote Cast Return Code generation public key (Kc_{id}^4)

If the validation of some of these proofs fails, the process is stopped. Otherwise, the configuration process continues as expected.

4.7 Create printing information

The information to be printed in each of the voting cards that will be provided to the voters before starting the election, is the following:

- Verification Card Codes (see Table 23 - Verification Card Codes).
- Ballot (see Table 13 - Ballot).
- Start Voting Key (SVK_{id}).

4.7.1 Printing Information if extended authentication is used

If the extended authentication is used, the Voter Alias are included as part of the information to be printed in each voting card. In addition, the Start Voting Key (SVK_{id}) is substituted by the Authentication Key.

4.8 Generation of Vote Verification Context Information

It generates the set of configuration information related to a Verification Card Set that should be stored in the Vote Verification Context. The set of information is grouped as follows:

- Verification Card Data (see Table 22 - Verification Card Data).
- Verification Card Set Control Components Data:

Verification Card Set Control Components Data

- Election Event ID ($\boldsymbol{eeid}$)
- Verification Card Set ID ($\boldsymbol{vcs_{id}}$)
- List of:
  - Verification Card ID ($\boldsymbol{vc_{id}}$)
  - $\boldsymbol{CCR_1}$ Voter Choice Return Code generation public key ($\boldsymbol{K_{id}^1}$)
  - $\boldsymbol{CCR_2}$ Voter Choice Return Code generation public key ($\boldsymbol{K_{id}^2}$)
  - $\boldsymbol{CCR_3}$ Voter Choice Return Code generation public key ($\boldsymbol{K_{id}^3}$)
  - $\boldsymbol{CCR_4}$ Voter Choice Return Code generation public key ($\boldsymbol{K_{id}^4}$)
  - $\boldsymbol{CCR_1}$ Voter Vote Cast Return Code generation public key ($\boldsymbol{Kc_{id}^1}$)
  - $\boldsymbol{CCR_2}$ Voter Vote Cast Return Code generation public key ($\boldsymbol{Kc_{id}^2}$)
  - $\boldsymbol{CCR_3}$ Voter Vote Cast Return Code generation public key ($\boldsymbol{Kc_{id}^3}$)
  - $\boldsymbol{CCR_4}$ Voter Vote Cast Return Code generation public key ($\boldsymbol{Kc_{id}^4}$)

Table 24 - Verification Card Set Control Component Data

- Verification Card Set Data:

Verification Card Set Data
- Election Event ID ($eeid$)
- Verification Card Set ID (vcs_{id})
- Choice Return Codes Encryption public key (pk_{CCR})
- Verification Card Set Issuer Certificate
- Vote Cast Return Code Signer Certificate

Table 25 - Verification Card Set Data

- Codes Mapping Table Context Data:

Codes Mapping Table Context Data
- Set of (Verification Card ID (vc_{id}) – Codes Mapping Tables)

Table 26 - Codes Mapping Table Context Data

- Vote Verification Context Data:

Vote Verification Context Data
- Election Event ID ($eeid$)
- Verification Card Set ID (vcs_{id})
- Signed CCR_1 Choice Return Codes Encryption public key (pk_{CCR_1})
- Signed CCR_2 Choice Return Codes Encryption public key (pk_{CCR_2})
- Signed CCR_3 Choice Return Codes Encryption public key (pk_{CCR_3})
- Signed CCR_4 Choice Return Codes Encryption public key (pk_{CCR_4})
- Codes Secret Key KeyStore
- Codes Secret Key password (encrypted with the Vote Verification Context System public key)
- Encryption parameters
- Election Key ID

Table 27 - Vote Verification Context Data

4.9 Generation of Voter Materials Context Information

It generates the set of configuration information related to a Voting Card Set that should be stored in the Voter Material Context. The set of information is the following:

- Voter Information: list of resource identifiers that will be assigned to a voter using a specific Voting Card

Voter Information
Contains one row per Verification Card ID (vc_{id}) with the following information: - Election Event ID ($eeid$) - Ballot ID (bid) - Ballot Box ID ($bbid$) - Voting Card Set ID ($vcds_{id}$) - Voting Card ID (vcd_{id}) - Credential ID (c_{id}) - Verification Card Set ID (vcs_{id}) - Verification Card ID (vc_{id})

Table 28 - Voter Information

- Credential Data: credential data associated to each Voting Card, to be provided to the voter during the authentication phase (see Table 21 - Credential Data)

4.10 Generation of Extended Authentication Context Information

Generates the set of configuration data related to every Voting Card that should be stored in the Extended Authentication Context.

Extended Authentication Information
- Authentication ID - Election Event ID ($eeid$) - Credential ID (c_{id}) - PBKDF2(ChallengeAnswer, Salt) - Salt - Encrypted Start Voting Key

Table 29 - Extended Authentication Data

4.11 Password protection

The election KeyStores password should be encrypted using the corresponding system context public keys. Before starting the election, the system context KeyStores passwords will be manually introduced or requested to the password manager, to open the KeyStores that contains the private keys to decrypt the election KeyStores passwords.

- Ballot box KeyStore password encrypted with the Tenant Election Information Context System public key (TEI_{pk}).
- Authentication Token Signer KeyStore password encrypted with the Tenant Authentication Context System public key (TAC_{pk}).

- Choice Return Codes Encryption KeyStore password encrypted with the Tenant Vote Verification Context System public key (TVV_{pk}).
- Code Secret key KeyStore encrypted with the Tenant Vote Verification Context System public key (TVV_{pk}).

4.12 Administration Board signature at configuration

4.12.1 Administration Board private key reconstruction

The members of the Administration Board enter their smartcards in the computer where the SDM is executed (e.g., Print Office or Canton environment). Only the threshold number of shares is required to reconstruct the key, although more shares can be used. To compute the reconstruction of the key, the Shamir Threshold Secret Sharing reconstruction algorithm is called.

For each share, its digital signature is verified using the Administration Board public key (AB_{pk}) in the X.509 certificate.

- **Certificate validation**: In case the X.509 certificate of the AB had to be previously downloaded from the Administration Portal, the chain of certificates should be verified up to the root.

Then, it should be verified that the reconstructed private key is correct:

- **Private key verification:** Call the Digital signature generation primitive to sign a test message with the reconstructed key and verify it using the AB certificate. If the validation is correct, the reconstructed key is also correct.

4.12.2 Data to sign

The following data packs should be signed by the Administration Board using the Digital signature generation primitive before uploading it to the corresponding online voting platform component. A precondition for this step is that the Administration Board has been constituted.

- **Authentication Voter Data** (see Table 9- Authentication Voter Data)
- **Authentication Context Data** (see Table 10 - Authentication Context Data)
- **Election Information Context Data** (see Table 11 - Election Information Context Data)
- **Voting Workflow Context Data** (see Table 12 - Voting Workflow Context Data)

For each Ballot Box one signature of:

- **Ballot Box Information** (see Table 14 - Ballot Box Information)
- **Ballot Box Context Data** (see Table 15 - Ballot Box Context Data)
- **Ballot Box Voter Data** (see Table 16 - Ballot Box Voter Data)

For each Verification Card Set ID (vcs_{id}), one signature of:

- **Verification Card Data** (see Table 22 - Verification Card Data)
- **Verification Card Set Data** (see Table 25 - Verification Card Set Data)
- **Verification Card Set Control Components Data** (see Table 24 - Verification Card Set Control Component Data)
- **Codes Mapping Tables Context Data** (see Table 26 - Codes Mapping Table Context Data)
- **Vote Verification Context Data** (see Table 27 - Vote Verification Context Data)

For each Ballot ID (bid), one signature of the **ballot** (see Table 13 - Ballot).

For each Voting Card Set ID $(vcds_{id})$, one signature of:

- **Voter Information** (see Table 28 - Voter Information)
- **Credential Data** (see Table 21 - Credential Data)
- **Extended Authentication Data** (see Table 29 - Extended Authentication Data)

For each electoral authority one signature of:

- **Electoral Authority Data** (see Table 17 - Electoral Authority Data)
- **Election Key Data** (see Table 18 - Election Key Data)

4.13 Administration Board signature verification at configuration

When the configuration information is uploaded to a context / service, its signatures should be verified, using the Administration Board certificate previously uploaded to the Certificate Registry. The upload is successful only if the signatures are verified.

The configuration information needed to verify the AB signature should be stored in the context/service, so that it can be verified in future (not just during upload).

Note: It should be verified that the configuration has been signed by an AB which is entitled to do so, issued by the Tenant for which the election event has been created.

5 Voting phase

This phase starts when the voter enters the Start Voting Key (SVK_{id}) into the application. Then the following steps are executed:

- Protocol GetID algorithm: Obtains the voter's identifier (Credential ID (c_{id})) from the Start Voting Key (SVK_{id}).
- Authentication: The voter is authenticated in the system using the challenge-response mechanism. At the end of the process, the voter receives an Authentication Token that is

included and validated in every request that the voting client sends to the voting server. They also receive all the necessary information to cast their vote.

- Protocol GetKey algorithm: Obtains the KeyStore password ($KSkey_{id}$) from the Start Voting Key (SVK_{id}) to retrieve the Verification Card private key (k_{id})
- Send a vote:
    - Protocol CreateVote algorithm: The voters select their voting options and the voting client encrypts them using the Election public key (EL_{pk}). Partial Choice Return Codes are computed using the Verification Card private key (k_{id}) and they are also encrypted. Cryptographic proofs linking the contents of both ciphertexts are generated. The whole vote is signed and sent to the voting server.
    - Protocol ProcessVote algorithm: The vote is validated both by the voting server and by the Control Components.
    - Protocol CreateCC algorithm: The pre-Choice Return Codes are computed after the interaction between the Voting Server and the Control Components. This protocol defines which are the operations done by these system components.
    - Protocol CreateRC algorithm: The short Choice Return Codes are retrieved from the mapping table given the pre-Choice Return Codes computed by the previous algorithm.
    - Generate receipt and store the vote: If the short Choice Return Codes are correctly retrieved, they are sent to the voter, the receipt is generated, and the vote is stored in the Ballot Box. The voter checks that the short Choice Return Codes received correspond with the options selected.
- Protocol GetCC algorithm: In case the voter logs out after sending the vote and then logs in back, the system returns the short Choice Return Codes.
- Confirm a vote:
    - Protocol Confirm algorithm: The Confirmation Message is computed from the Ballot Casting Key (BCK^{id}) introduced by the voters to confirm their vote. The information is signed and sent to the voting server.
    - Protocol ProcessConfirm algorithm: The Confirmation Message is validated by the voting server and the short Vote Cast Return Code is retrieved from the mapping table after the interaction among the voting server and the Control Components. If the short Vote Cast Return Code is correctly retrieved, it is sent to the voter together with the vote and the receipt.
- Client-side receipt validation: The voting client validates the receipt and the voter checks that the short Vote Cast Return Code received corresponds with that code in the Voting Card.

In case any step of the processes described in this section fails due to a validation failure, or to a request for a non-existing resource, the process stops, the error is logged and forwarded to the voter. An error message is sent to the client context (in the event the error happened in the server-side), which shows an error message in the screen. The voter will then have to log out and start the process again from the last successful point:

- In case the vote was stored successfully, and the Choice Return Codes sent to the client, the voter starts in the confirmation page where Choice Return Codes are shown on the screen after login.
- In case the vote and the confirmation were stored successfully, the voter starts in the last page where the Vote Cast Return Code and optionally the receipt and the signature are shown after login.
- In case an error was found in the vote structure after the vote was stored, and the Choice Return Codes were not successfully recovered, the voter will be blocked, and an error will be displayed.

5.1 Protocol GetID algorithm

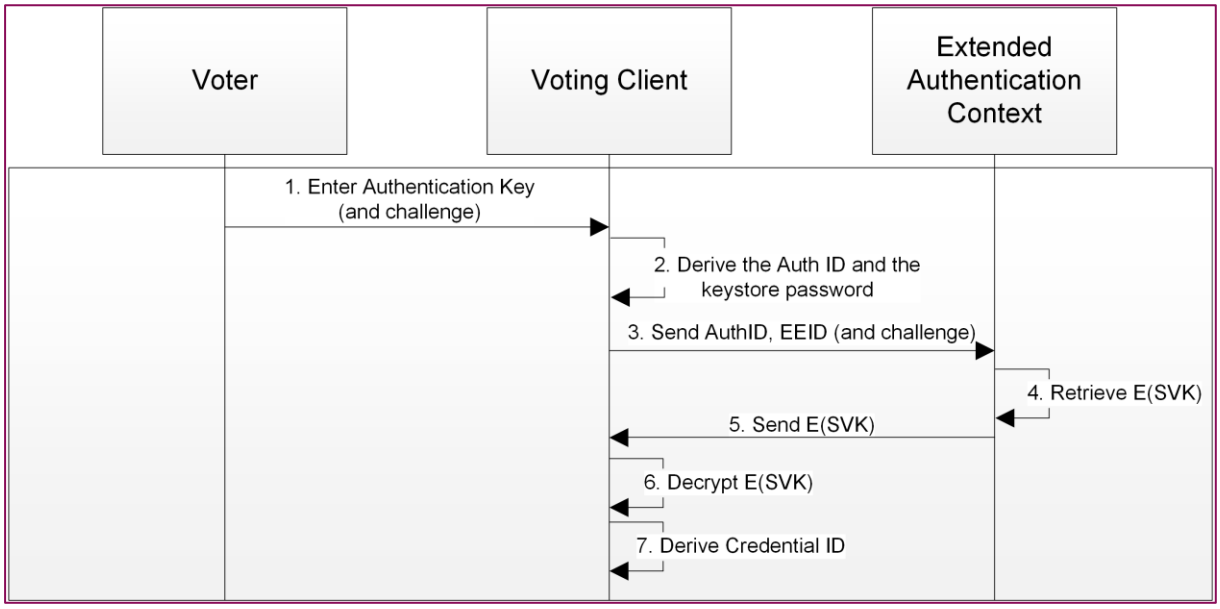

Figure 16 - Protocol GetID

1) The voter enters the Authentication Key (Start Voting Key (SVK_{id}) or a list of secrets) and an optional challenge into the application. The voter has a limited number of attempts (the maximum number of attempts is 5) to introduce the answer to the challenge. If the maximum number is reached, the voting card is blocked.

2) From the Authentication Key entered, the **Client Context**:

    a) Calls the Password-based key derivation function with the following inputs:

        - Password: Authentication Key value

- Salt: concatenation of the string ‘authID’ and the Election Event ID $(eeid)$

The result is the Auth ID.

b) Calls the Password-based key derivation function with the following inputs:

- Password: Authentication Key value
- Salt: concatenation of the string ‘authPassword’ and the Election Event ID $(eeid)$

The result is the KeyStore password.

3) Sends the Auth ID, the Election Event ID $(eeid)$ and the challenge (if needed) to the server-side.

4) The **Extended Authentication Context** receives the Auth ID, the Election Event ID $(eeid)$, the Tenant ID and the challenge (if needed) and does the following actions:

a) Retrieves the extended authentication information (AuthenticationID, PBKDF2(ChallengeAnswer, Salt), Salt, Encrypted(SVK^{id})) using the Auth ID, the Election Event ID $(eeid)$ and the Tenant ID.

b) Checks that the number of attempts does not exceed the maximum number of attempts allowed. When the value reaches the maximum, the entry for the Auth ID is set to BLOCKED.

c) Calls the Password-based key derivation function with the following inputs:

- Password: Challenge Answer value.
- Salt: salt stored in the extended authentication information.

d) Compares the computed value with the PBKDF2(ChallengeAnswer, Salt) retrieved. If the comparison is ‘false’, increase the number of attempts.

5) The **Extended Authentication Context** sends the Encrypted (SVK^{id})) to the Client Context.

6) The **Client Context** calls the Symmetric decryption primitive to decrypt the encrypted SVK using the KeyStore password derived from the Authentication Key.

7) The **Client Context** calls the Password-based key derivation function with the following inputs:

a) Password: Start Voting Key (SVK_{id}).

b) Salt: concatenation of the string ‘credentialid’ and the Election Event ID $(eeid)$.

and obtains the Credential ID (c_{id}).

5.2 Authentication

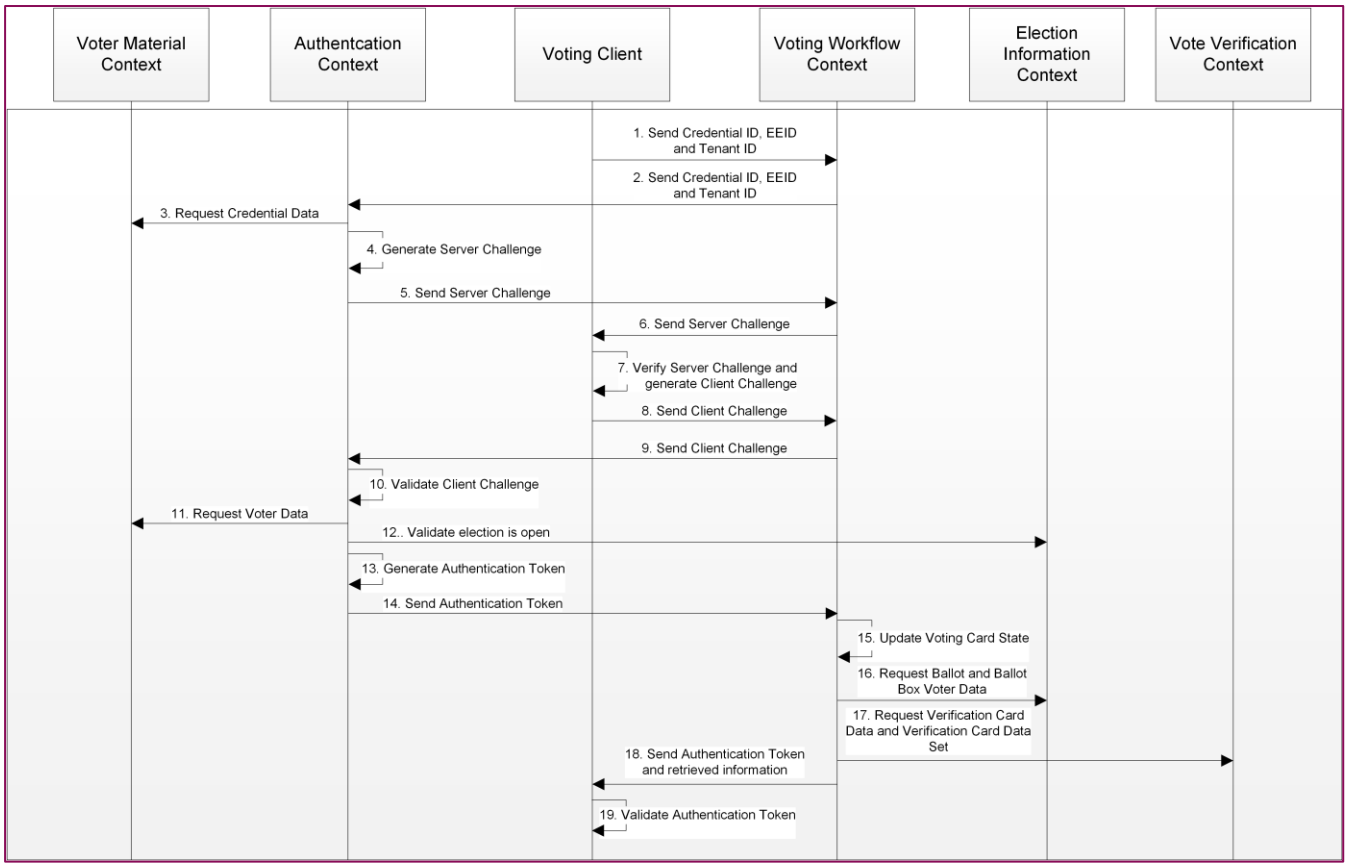

Figure 17 - Authentication

In this phase, the Client Context and the Authentication Context engage in an authentication protocol. After that, the Authentication Context will issue an Authentication Token to the voter and provide the information to the Client Context necessary for performing the following steps of the protocol, depending on the status of the voter in the system. We divide the authentication process in two steps:

- Challenge-response mechanism.
- Authentication token generation.

5.2.1 Challenge-response mechanism

1) The **Client Context** sends the Credential ID (c_{id}), the Election Event ID ($eeid$) and the Tenant ID to the **Voting Workflow Context.**

2) The **Voting Workflow Context** receives the Credential ID (c_{id}), the Election Event ID ($eeid$) and the Tenant ID and sends them to the **Authentication Context.**

3) The **Authentication Context** receives the Credential ID (c_{id}), the Election Event ID ($eeid$) and the Tenant ID and asks for the corresponding Credential Data to the **Voter Material Context** using the Credential ID (c_{id}), the Election Event ID ($eeid$) and the Tenant ID. In case the Credential Data entry is not present, an error message is created and forwarded to the client

context, which shows an error message to the voter. The voter is instructed to restart the process.

4) Once the **Authentication Context** has received the Credential Data, it generates a Server Challenge Message in the following way:
    a) Calls the Random value generation primitive to generate a value of 16 bytes.
    b) Generates a timestamp with the current time.
    c) Calls the Digital signature generation primitive to sign the timestamp, the server random value, the Credential ID (c_{id}) and the Election Event ID ($eeid$) using the Authentication Token Signer private key (ATs_{sk}).
    d) The Server Challenge Message is composed by the server timestamp, the server random and the signature in base64 format.
5) The Server Challenge Message, the Credential Data and the set of Certificates corresponding to that Election Event ID ($eeid$) is sent to the **Voting Workflow Context**.
6) The **Voting Workflow Context** sends the information to the **Client Context.**
7) The **Client Context**:
    a) Verifies the certificate validity and the certificate chains of the certificates received:
        - [Election Event Root CA]
        - [Authentication Token Signer, Services CA, Election Event Root CA]
        - [Administration Board Certificate, Tenant CA, Platform Root CA]
        - [Credentials CA, Election Event Root CA]
        - [Credential ID Auth, Credentials CA, Election Event Root CA]
        - [Credential ID Sign, Credentials CA, Election Event Root CA]
    b) Validates the signature over the Server Challenge Message values (server random, server timestamp, Credential ID (c_{id}), Election Event ID ($eeid$)), using the Authentication Token Signer public key (ATs_{pk}) from the corresponding certificate in the set received.
    c) Check that the Credential ID (c_{id}) from the Server Challenge Message matches the one in Credential Data.
    d) Calls the Password-based key derivation function with the following information:
        - Password: Start Voting Key (SVK_{id})
        - Salt: concatenation of the string ‘KeyStorepin’ and the Election Event ID ($eeid$)

    The result is the KeyStore symmetric encryption key ($KSkey_{id}$).

e) Opens the KeyStore from Credential Data using the KeyStore password ($KSkey_{id}$), retrieving:

- The Credential ID authentication ($k^{a}_{C_{id}}$) and Credential ID signing ($k^{s}_{C_{id}}$) private keys.
- The X.509 certificates associated and their certificate chains.

f) Generates a Client Challenge Message in the following way:

- It calls the Random value generation primitive to generate a value of 16 bytes.
- It signs it together with the Server Challenge signature using the Credential ID authentication private key ($k^{a}_{C_{id}}$) and calling the Digital signature generation primitive.

8) Sends the Client Challenge Message, the Credential ID Authentication certificate, the Credential ID (c_{id}), the Election Event ID ($eeid$) and the Tenant ID to the **Voting Workflow Context.**

9) The **Voting Workflow Context** sends the information to the **Authentication Context.**

10) The **Authentication Context** verifies the information sent by the Client Context:

a) Validates the certificate chain (Credential ID Auth Certificate, Credentials CA Certificate, Election Event Root CA).

b) Validates the signature over the Server Challenge Message values (server random, timestamps, Credential ID (c_{id}) and Election Event ID ($eeid$)) using the Authentication Token Signer public key (ATs_{pk}).

c) Verifies that the difference between the current time and the timestamp in the *Server Challenge Message* is not greater than the challenge-response expiration time (already set in the context).

d) Validates that the Credential ID (c_{id}) of the request matches the Credential ID (c_{id}) contained in the Credential ID Authentication Certificate.

e) Validates the signature over the Client Challenge Message values (client random value, server challenge signature), using the public key in the Credential ID Authentication Certificate.

5.2.2 Authentication Token generation

11) In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the login process. If not, the **Authentication Context** asks for voter-related data to the **Voter Material Context.**

12) The **Authentication Context** validates if the election is still open asking the **Election Information Context**.

13) If the validation is successful, the **Authentication Context** prepares the Authentication Token:

    a) Calls the Random value generation to generate the Authentication Token ID as a value of 16 bytes.

    b) Generates a timestamp with the current time.

    c) Generates an Authentication Token containing the Voter Information, a timestamp and the Authentication Token ID.

    d) Calls the Digital signature generation primitive to sign the Authentication Token using the Authentication Token Signer private key (ATs_{sk}).

14) The **Authentication Context** sends the Authentication Token back to the **Voting Workflow Context**.

15) The **Voting Workflow Context** using the Voting Card ID (vcd_{id}) in the Authentication Token, retrieves the voting card state and initializes it.

16) Using the Ballot ID (bid) and the Voting Card ID (vcd_{id}) in the Authentication Token, and the Election Event ID ($eeid$) and the Tenant ID from the request, the **Voting Workflow Context** retrieves the Ballot, the ballot text and the Ballot Box Voter Data with their corresponding signatures from the **Election Information Context.**

17) Using the Verification Card ID (vc_{id}) and the Voting Card ID (vcd_{id}) in the Authentication Token, and the Election Event ID ($eeid$) and the Tenant ID from the request, the **Voting Workflow Context** retrieves the Verification Card Data (Verification Card KeyStore and Signed verification card public key), the Verification Card Set Data (Choice Return Codes Encryption public key (pk_{CCR}), Verification Card Set Issuer Certificate, Vote Cast Return Code Signer Certificate) and their signatures from the **Vote Verification Context**.

18) The **Voting Workflow Context** sends the Authentication Token, the Ballot, the Ballot Texts, the Ballot Box Voter Data, the Verification Card Data, the Verification Card Set Data and their signatures, to the **Client Context**.

19) The **Client Context:**

    a) Signer certificate.

    b) Checks that the fields Election Event ID ($eeid$), Credential ID (c_{id}), Ballot ID (bid), Ballot Box ID ($bbid$) and Verification Card ID (vc_{id}) in the Authentication Token are consistent with the other data already known or received.

c) Checks that the fields Ballot ID (bid), Ballot Box ID $(bbid)$ and Verification Card ID (vc_{id}) in the Authentication Token are consistent with the other data already known or received (i.e., that the Ballot received has the same Ballot ID (bid)).

d) Checks whether the Ballot ID (bid) of the received Ballot is the same as that in the Ballot Box Voter Data.

e) In case any of the previous validations fail, the application stops, and an error message is shown to the voter. The voter needs to log out and start the logging process again.

5.3 Protocol GetKey algorithm

1) Calls the Password-based key derivation function with the following information:

   a) Password: Start Voting Key (SVK_{id}).

   b) Salt: Concatenation of the string 'keystorepin' and the Election Event ID $(eeid)$.

   The result is the KeyStore symmetric encryption key $(KSkey_{id})$.

2) Uses the KeyStore symmetric encryption key $(KSkey_{id})$ KeyStoreto open the KeyStore in the Verification Card Data and recover the Verification Card private key (k_{id}).

5.4 Send a vote

5.4.1 Protocol CreateVote algorithm

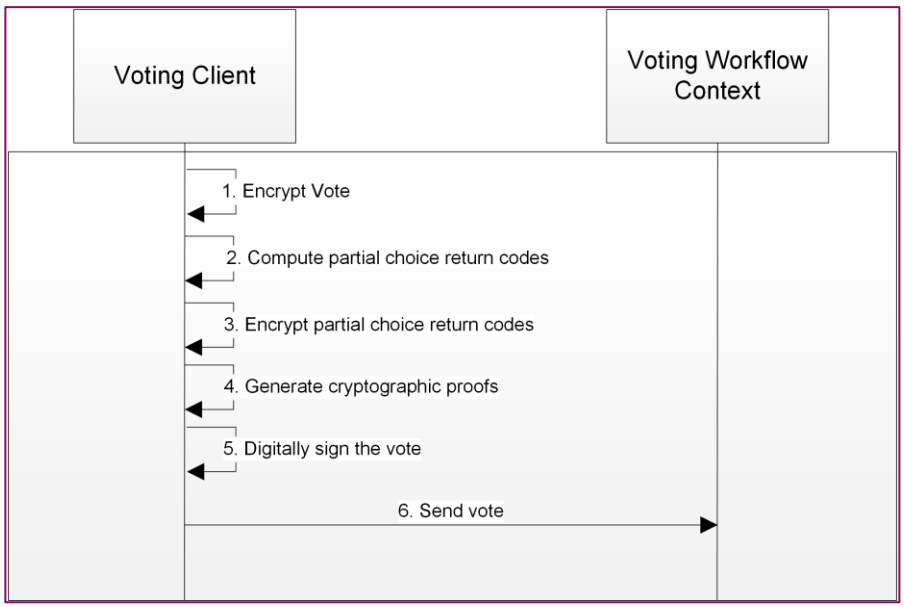

Figure 18 - Protocol CreateVote

After being authenticated, the ballot is presented to the voter, who selects the desired voting options. Then, the following steps are executed:

1) Compress the prime numbers received (which represent the selected voting options) by multiplying them together. The result will be encrypted using one of the components of the election public key contained in the Ballot Box Voter Data. If write-ins are allowed:

   a) The prime numbers associated with the write-ins that have been filled by the voter, are compressed together with the other received primer numbers.

      Note: The primer numbers associated with the write-in fields only indicates that the field has been filled in but does not give any information about the text inside the write-in.

   b) The content of each write-in field will be encrypted using one component of the Election Key (different from the one used to encrypt the product of primer numbers). Before encrypting the content should be translated into a number of the given mathematical groups. For this, one proposal is to:

      - transform the write-in field text (in UTF-8) to a fixed length number encoding,
      - check that this value is not bigger than q from the encryption parameters, and
      - raise the value to 2 and compute $mod\ p$. This will be the value provided to the encryption.

   Note: This specification assumes each write-in can be encoded as a numeric value smaller than 2047 bits.

   On the other hand, regarding the number of elements of the election public key, the only limit we know so far is the smartcard storage which forces us to use 60 maximum write-ins. The protocol is not limiting the number of write-ins at all.

Call the ElGamal encryption primitive[3] with input the compressed prime numbers, the write-ins $(w_1, \dots, w_k)$ and the Election public key $(EL_{pk}^{(1)}, \dots, EL_{pk}^{(m)})$. Obtain the following ciphertext:

$$E_1 = (g^r, \left(EL_{pk}^{(1)}\right)^r \cdot \prod_{i=1}^{\psi} v_i\,, \left(EL_{pk}^{(2)}\right)^r \cdot w_1, \left(EL_{pk}^{(3)}\right)^r \cdot w_2, \dots, \left(EL_{pk}^{(m)}\right)^r \cdot w_k)$$

If write-ins are allowed, the ciphertext E_1 must have the same number of elements (m) for every voter whether they have filled in the write-ins or not. In case the voter has not filled the write-in field, the string "2" is encrypted.

2) For each one of the voting options $\{v_i\}_{i=1}^{\psi}$, compute a partial Choice Return Code pCC_i^{id} using the Verification Card private key (k_{id}) of the voter represented by the Verification Card ID (vc_{id}):

$$pCC_i^{id} = v_i^{k_{id}}\ mod\ p \text{ (where } p \text{ is taken from the encryption parameters)}$$

No partial Choice Return Codes are generated for the write-in contents.

[3] If the election key contains more elements than the number of options to be encrypted, the unused elements are multiplied and used as the last element of the key.

3) Call the ElGamal encryption primitive to encrypt all the partial Choice Return Codes with the Choice Return Codes Encryption public key (pk_{CCR}) contained in the received Verification Card Set Data.

$$E_2 = (g^{r'}, \left(pk_{CCR}^{(1)}\right)^{r'} \cdot pCC_1^{id}, \left(pk_{CCR}^{(2)}\right)^{r'} \cdot pCC_2^{id}, \dots, \left(pk_{CCR}^{(\psi)}\right)^{r'} \cdot pCC_\psi^{id})$$

The vote sent must include, for each encrypted partial Choice Return Code, the correctness ID corresponding with that voting option with the flag "correctness = true".

4) Generate cryptographic proofs linking the encrypted partial Choice Return Codes and the encrypted product of primes:

   a) Call the Schnorr proof generation primitive with the following inputs:

      - Base element (group element): g
      - Exponent: r
      - Public input (group element): g^r
      - Additional information: "$SchnorrProof: Voter\ ID =$ " concatenated with the value of the Voting Card ID (vcd_{id}) concatenated with "$Election\ Event\ ID =$ " concatenated with the value of the Election Event ID ($eeid$).
      - Mathematical group (p, q, g)

      The result is the Schnorr proof (π_{sch}).

   b) Call the Exponentiation proof generation primitive with the following inputs:

      - Base elements (group elements): $[g, g^r, \left(EL_{pk_1}\right)^r \cdot \prod_{i=1}^{\psi} v_i]$
      - Exponents: $[k_{id}]$
      - Public input (group elements): $[K_{id}, (g^r)^{k_{id}}, \left(\left(EL_{pk_1}\right)^r \cdot \prod_{i=1}^{\psi} v_i\right)^{k_{id}}]$
      - Additional information "$ExponentiationProof$"
      - Mathematical group (p, q, g)

      The result is the Exponentiation proof (π_{exp}).

   c) Call the Plaintext Equality proof generation primitive with the following inputs:

      - Primary Ciphertext: $[(g^r)^{k_{id}}, \left(\left(EL_{pk_1}\right)^r \cdot \prod_{i=1}^{\psi} v_i\right)^{k_{id}}]$.
      - Primary public key: $[EL_{pk_1}]$
      - Primary randomness: $[r \cdot k_{id}]$

- Secondary Ciphertext: $[g^{r'}, \prod_{i=1}^{\psi}(pk_{CCR}^{(i)})^{r'} \cdot pCC_i^{id})$
- Secondary public key: $[\prod_{i=1}^{\psi} pk_{CCR}^{(i)}]$
- Secondary randomness: $[r']$
- Additional information "$PlaintextEqualityProof$"
- Mathematical group: (p, q, g)

The result is the Plaintext Equality proof $(\pi_{pleqenc})$.

5) Call the Digital signature generation primitive to sign the following information using the Credential ID signing private key ($k_{C_{id}}^{S}$):
   a) The encrypted vote (E_1):
      - The encrypted compressed primes
      - The encrypted write-ins value
   b) List of correctness IDs
   c) The Schnorr proof
   d) The Verification Card public key signature
   e) The Authentication Token signature
   f) The Voting Card ID (vcd_{id}) and the Election Event ID ($eeid$).
6) The following information is sent to the **Voting Workflow Context**:
   a) The encrypted vote (E_1)
   b) The encrypted partial Choice Return Codes (E_2)
   c) List of correctness IDs
   d) The Verification Card public key (K_{id})
   e) The Verification Card public key signature
   f) Digitally signed vote
   g) The Credential ID signing certificate
   h) The Credential ID (c_{id})
   i) Cryptographic proofs $(\pi_{sch}, \pi_{exp}, \pi_{pleqenc})$
   j) Ciphertext exponentiations
   k) The Authentication Token

5.4.2 Protocol ProcessVote algorithm

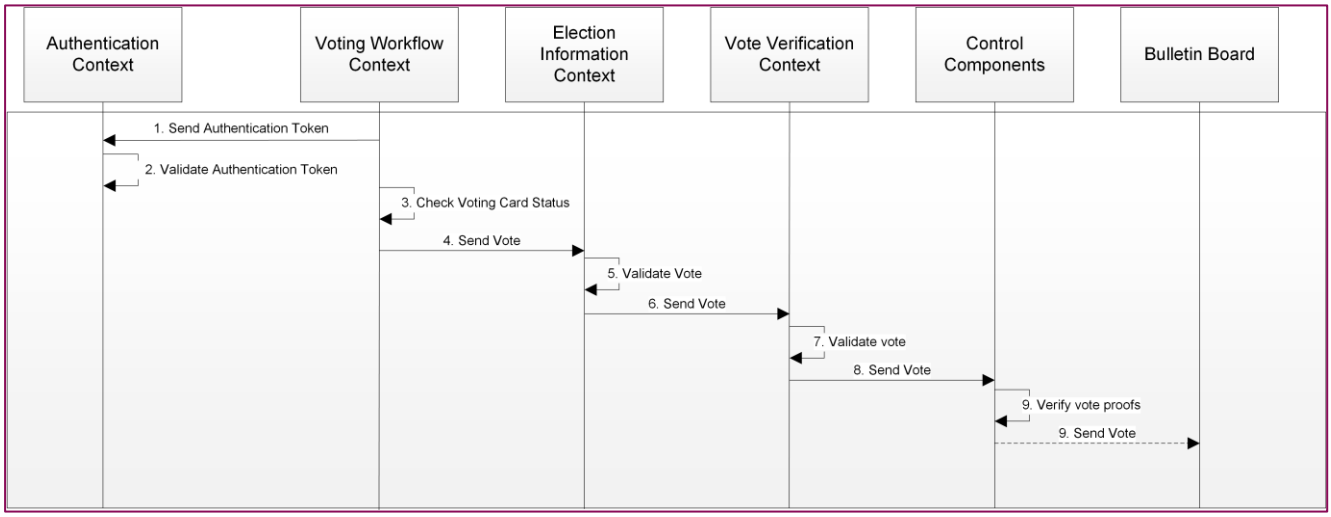

Figure 19 - Protocol ProcessVote

The following actions happen at the server-side, when a vote is received:

1) The **Voting Workflow Context** sends the Authentication Token to the Authentication Context to be validated.
2) The **Authentication Context:**
    a) Validates the signature of the Authentication Token, using the Authentication Token Signer certificate.
    b) Checks that the fields Tenant ID, Election Event ID ($eeid$), Voting Card ID (vcd_{id}), in the Authentication Token are consistent with the Voting Card ID (vcd_{id}) and Election Event ID ($eeid$) provided.
    c) Verifies that the Authentication Token has not expired.

    In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the login process.
3) If the validations of the Authentication Token are successful, the **Voting Workflow Context** checks the status of the Voting Card.
4) Only if the Voting Card status is NOT SENT, the vote is sent to the **Election Information Context** to be validated.
5) The **Election Information Context:**
    a) Validates that the Ballot Box is not blocked.
    b) Verifies if the election is out of period checking the current timestamp against the election dates from the election event configuration.

c) Performs the following validations over the Vote:

i. Checks that the Credential ID signing certificate (which is provided together with the Encrypted Vote) is issued for the same Credential ID (c_{id}) which is provided as a parameter.

ii. Validates the certificate chain: [Credential ID signing Certificate, Credentials CA, Election Event Root CA]

iii. Verifies the digital signature over the values in the vote, the Authentication Token signature, the Verification Card public key signature, the Voting Card ID (vcd_{id}) and the Election Event ID ($eeid$), using the Credential ID X.509 signing certificate (which is provided together with the Encrypted Vote).

iv. Checks that the Election Event ID ($eeid$), Tenant ID, Voting Card ID (vcd_{id}), Ballot Box ID ($bbid$), Ballot ID (bid) and Credential ID (c_{id}), inside the vote match the corresponding IDs in the request and in the vote.

v. Checks that the Credential ID (c_{id}) who signs the vote is the same Credential ID (c_{id}) from the request.

vi. Checks that the ciphertext elements are group elements.

vii. Checks that the number of votes cast by the voter does not exceed the maximum number of allowed votes per voting card and authentication token.

viii. Verifies that the encrypted vote contains the correct number of elements: number of write-ins that can be filled in the ballot (the number of voting options with the label “writein”) plus 2 (the first element of the ciphertext and the encrypted compressed primes).

ix. Validates the number of elements of the encrypted partial Choice Return Codes, against what is defined by the election configuration, using the attributes sent with the vote.

x. Uses the function encryptedCorrectnessRule defined in the ballot to validate that the number of selected options, including blank selections, is equal to the maximum established.

xi. Verifies that a vote is not already stored for that Voting Card ID (vcd_{id}).

If all the validations are successful, a hash of the vote is stored to verify later (after the Choice Return Codes computation) that the vote that is going to be saved in the Ballot Box was correctly validated by the system. In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the login process.

6) If the validations performed in the Election Information Context are successful, the vote is sent to the **Vote Verification Context.**

7) The **Vote Verification Context:**

    a) Verifies the digital signature of the Verification Card public key (K_{id}) using the Verification Card Set Issuer certificate.

8) In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the logging process. If validations are successful, the Vote Verification Context broadcasts the information to every Control Component (CCR_j).

9) Each Control Component (CCR_j) does the following actions:

    a) Checks if the voter has already sent a vote. If this is the case, the Control Component stops the process, logs an error and forwards it to the client context, which shows an error message to the voter. In this way, we prevent the Control Components from computing more than once the Choice Return Codes for that voter and we avoid an attack coming from the server.

    b) Verifies the Schnorr proof (π_{sch}), the Exponentiation proof (π_{exp}) and the Plaintext Equality proof ($\pi_{pleqenc}$). In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the logging process.

    c) Logs the vote (only the fields included in the signature) and its signature in a format that the signature could be validated by an external process.

5.4.3 Protocol CreateCC algorithm

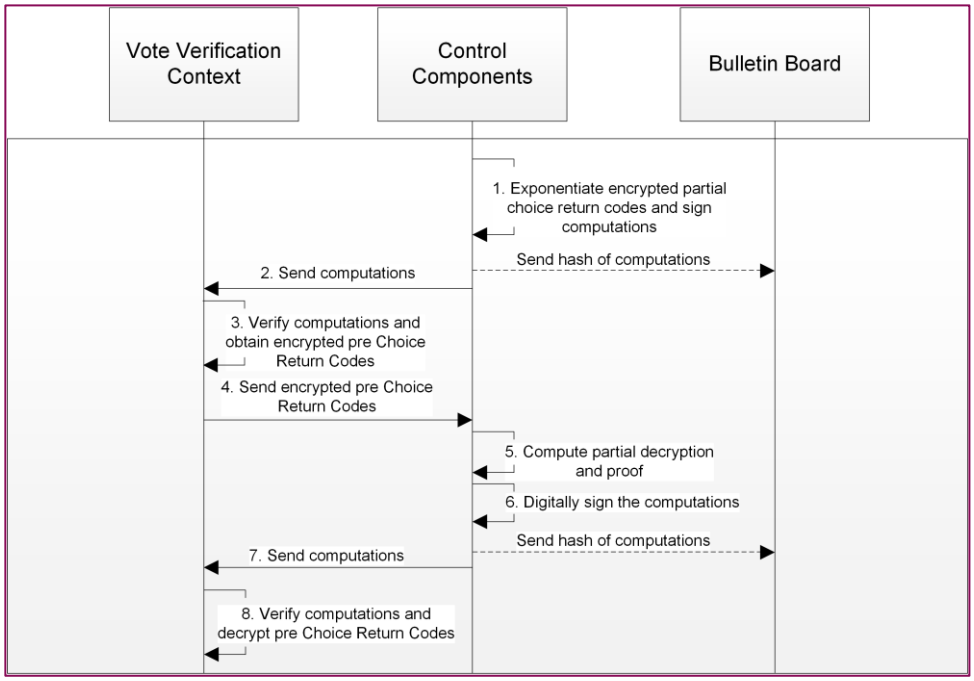

Figure 20 - Protocol CreateCC

The Choice Return Codes are generated between the Vote Verification Context and the Control Components:

1) Each Control Component (CCR_j) does the following actions:

   a) Derives the Voter choice return code generation private key (k_{id}^j) from the corresponding CCR_j choice return code generation private key (k_j') and the Verification Card ID (vc_{id}):

      i. Computes $k_{id}^j = KDF(vc_{id}||k_j',\ p\ length)$ calling the Key Derivation Function: KDF1 specification primitive.

      ii. Truncates the result to have 2047 bits.

      iii. Checks that $1 \leq k_{id}^j \leq q - 1$

      iv. If the derived value is equal or greater than q, compute again a derivation but using as input the derived value $KDF(k_{id}^j,\ p\ length)$.

      v. k_{id}^j is the Vote choice return code generation private key.

   b) Computes the exponentiation:

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

and calls the Exponentiation proof generation primitive to compute a proof of knowledge of the exponent k_{id}^{j}. The inputs of the primitive are the following:

- Base elements (group elements): $[g, E_2]$
- Exponents: $[k_{id}^{j}]$
- Public input (group elements): $[K_{id}^{j}, {E_2}^{k_j^{id}}]$
- Additional information "$ExponentiationProof$"
- Mathematical group (p, q, g)

The result is $\sigma_{CCR_j}^{1}$.

c) Calls the Digital signature generation primitive to sign the exponentiated ciphertext ${E_2}^{k_{id}^{j}}$ and the $\sigma_{CCR_j}^{1}$ using the CCR_j signing private key $(sk_{CCR_j}^{s})$.

d) Stores the Verification Card ID (vc_{id}) in the database as an evidence that Choice Return Codes have been already computed for that voter.

2) Sends the signed information to the **Vote Verification Context.**

3) The **Vote Verification Context** receives the following signed information from each Control Component: $\{({E_2}^{k_{id}^{1}}, \sigma_{CCR_1}^{1}); ({E_2}^{k_{id}^{2}}, \sigma_{CCR_2}^{1}); ({E_2}^{k_{id}^{3}}, \sigma_{CCR_3}^{1}); ({E_2}^{k_{id}^{4}}, \sigma_{CCR_4}^{1})\}$ and:

a) Retrieves the signed Choice Return Code generation public keys $(K_{id}^{1}, K_{id}^{2}, K_{id}^{3}, K_{id}^{4})$ stored in Verification Card Set Control Components Data associated to the Verification Card ID vc_{id}.

b) Verifies the signatures and the proofs, multiplies the exponentiated ciphertexts ${E_2}^{k_{id}^{j}}$ and obtains the encrypted pre-Choice Return Codes

$$\prod_{j=1}^{4} {E_2}^{k_{id}^{j}} = \left(g^{\mathrm{r}'\cdot\hat{k}}, \left(pk_{CCR}^{(1)}\right)^{\mathrm{r}'\cdot\hat{k}} \cdot v_1^{k}, \ldots, \left(pk_{CCR}^{(\psi)}\right)^{\mathrm{r}'\cdot\hat{k}} \cdot v_\psi^{k}\right)$$

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

4) The **Vote Verification Context** sends $g^{\mathrm{r}'\cdot\hat{k}}$ to the control components to obtain the partial decryption.

5) Each Control Component receives $g^{\mathrm{r}'\cdot\hat{k}}$ from the **Vote Verification Context** and uses its private key share $\left(sk_{CCR_j}^{(1)}, \ldots, , sk_{CCR_j}^{(\psi)}\right)$ to compute the partial decryption:

$$\{g^{r'\cdot sk_{CCR_j}^{(1)}\cdot\hat{k}\cdot}, \ldots, g^{r'\cdot sk_{CCR_j}^{(\psi)}\cdot\hat{k}}\}.$$

They also compute a proof of knowledge $(\sigma_{CCR_j}^{2})$ of the private key $sk_{CCR_j}^{(1)}, \ldots, , sk_{CCR_j}^{(\psi)}$ but prior to the computation the following values are calculated:

a) Compressed CCR_j Choice Return Codes Encryption private key: $sk_{CCR_j}^{(1)} + \cdots + sk_{CCR_j}^{(\psi)}$

b) Compressed CCR_j Choice Return Codes Encryption public key: $pk_{CCR_j}^{(1)} \cdot pk_{CCR_j}^{(2)} \cdots pk_{CCR_j}^{(\psi)}$

c) Compressed exponentiated gammas: $g^{r' \cdot sk_{CCR_j}^{(1)} \cdot \hat{k} \cdot} \cdot g^{r' \cdot sk_{CCR_j}^{(2)} \cdot \hat{k} \cdot} \cdots g^{r' \cdot sk_{CCR_j}^{(\psi)} \cdot \hat{k}} = g^{r' \cdot \hat{k} \cdot (sk_{CCR_j}^{(1)} + \cdots + sk_{CCR_j}^{(\psi)})}$

Calls the Exponentiation proof generation primitive with the following inputs:

a) Base elements (group elements): $[g, g^{r' \cdot \hat{k}}]$

b) Exponents: $[CompressedSecretKey]$

c) Public input (group elements): $[CompressedPublicKey, CompressedExponentiatedGammas]$

d) Additional information "$ExponentiationProof$"

e) Mathematical group (p, q, g)

The result is $\sigma_{CCR_j}^2$.

6) Calls the Digital signature generation primitive to sign the partial decryption and the proof using the CCR_j signing private key ($sk_{CCR_j}^s$).

7) The signed information is sent to the **Vote Verification Context.**

8) The **Vote Verification Context** receives the following signed information from each Control Component: $\left\{g^{r' \cdot sk_{CCR_j}^{(1)} \cdot \hat{k} \cdot}, \ldots, g^{r' \cdot sk_{CCR_j}^{(\psi)} \cdot \hat{k}}\right\}, \sigma_{CCR_j}^1$ and does the following actions:

a) Retrieves the Signed CCR_j Choice Return Codes generation public keys ($g^{k'_1}, g^{k'_2}, g^{k'_3}, g^{k'_4}$) stored in the Verification Card Set Control Components Data to verify the proofs ($\sigma_{CCR_1}^1, \sigma_{CCR_2}^1, \sigma_{CCR_3}^1, \sigma_{CCR_4}^1$)

b) Verifies the signatures and the proofs.

c) Multiplies all the partial decryptions and obtains the decrypted pre-Choice Return Codes $\{pC_i^{id}\}_{i=1}^{\psi}$ for each voting option:

$$g^{-r' \cdot sk_{CCR}^{(1)} \cdot \hat{k}} = \prod_{j=1}^{4} g^{-r' \cdot sk_{CCR_j}^{(1)} \cdot \hat{k}}$$

$$\vdots$$

$$g^{-r' \cdot sk_{CCR}^{(\psi)} \cdot \hat{k}} = \prod_{j=1}^{4} g^{-r' \cdot sk_{CCR_j}^{(\psi)} \cdot \hat{k}}$$

Note that as $pk_{CCR}^{(1)} = g^{sk_{CCR}^{(1)}}, \dots, pk_{CCR}^{(\psi)} = g^{sk_{CCR}^{(\psi)}}$, if the server multiplies every ciphertext element by its corresponding partial decryption element, the pre-Choice Return Codes $\{pC_i^{id}\}_{i=1}^{\psi}$ are obtained:

$$pC_1^{id} = v_1^k = g^{-r' \cdot sk_{CCR}^{(1)} \cdot \hat{k}} \cdot \left(pk_{CCR}^{(1)}\right)^{r' \cdot \hat{k}} \cdot v_1^k$$

$$\vdots$$

$$pC_\psi^{id} = v_\psi^k = g^{-r' \cdot sk_{CCR}^{(\psi)} \cdot \hat{k}} \cdot \left(pk_{CCR}^{(\psi)}\right)^{r' \cdot \hat{k}} \cdot v_\psi^k$$

5.4.4 Protocol CreateRC algorithm

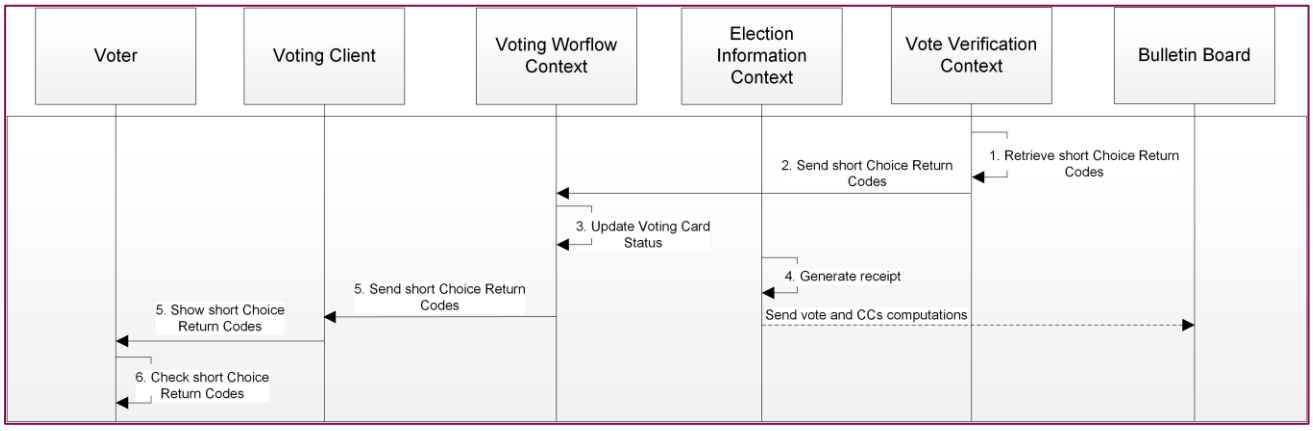

Figure 21 - Protocol CreateRC

Once the pre-Choice Return Codes are obtained, the **Vote Verification Context** does the following actions per Verification Card ID (vc_{id}):

1) Looks for the Codes Mapping Table corresponding to the Verification Card ID (vc_{id}) and Election Event ID ($eeid$) and for each pre-Choice Return Code $pC_i^{id} = v_i^k$:

   a) Concatenates it with the Verification Card ID (vc_{id}), the Election Event ID ($eeid$) and the corresponding voting option attributes with the flag “correctness = true”. Call the Hash generation primitive with input the concatenated value. The result is the long Choice Return Code:

$$lCC_i^{id} = Hash(v_i^k || vc_{id} || EEID || \{attributes\})$$

- For each long Choice return Code, call the Hash generation primitive to compute the hash of the long Choice Return Code (lCC_i^{id}) concantenated with the Codes Secret Key (C_{sk}) and call the Key Derivation Function: KDF1 specification to generate the Choice Return Code encryption symmetric key $skcc_i^{id} = KDF(Hash(lCC_i^{id}||C_{sk}), 256\ bits)$.
- Retrieves the encrypted short Choice Return Code from the mapping table using $Hash(lCC_i^{id})$ and calls the Symmetric decryption primitive to decrypt it using $skcc_i^{id}$. The result is the short Choice Return Code to be sent to the voter.

b) Checks that all the retrieved short Choice Return Codes are different. If not, interrupt the process and send a validation error.

2) The short Choice Return Codes and the computations done by the Control Components are sent to the **Voting Workflow Context.**

3) If the short Choice Return Codes are correctly retrieved, the **Voting Workflow Context** updates the status of the voting card to SENT BUT NOT CAST.

4) The **Election Information Context** stores the vote, the Control Components computations and generates the receipt (see next section).

5) The Choice Return Codes are sent back to the Voting Client, which displays them in the screen.

6) The voter checks, using his/her voting card, that the Choice Return Codes received are those corresponding with the voting options selected.

7) In case of any error in the validations/generation, Choice Return Codes do not have to be sent back to the Voting Client.

5.4.5 Generate receipt and store vote

The **Election Information Context** generates the receipt and stores the vote and receipt in the Ballot Box: It does the following:

1) Verifies that the vote was correctly validated checking if there is a hash of that vote stored.

2) Computes a Receipt from the vote: It calls the Hash generation primitive and computes a hash of the vote signature, the authentication token signature, the verification card public key signature, the Election Event ID ($eeid$) and the Voting Card ID (vcd_{id}).

3) Calls the Digital signature generation primitive to sign the Receipt using the Ballot Box Signer private key ($BBs_{sk}^{bb_{id}}$).

4) Checks that the vote that is going to be stored has been validated.

5) Stores the vote (Tenant ID, Election Event ID ($eeid$), Ballot ID (bid), Ballot Box ID ($bbid$), Voting Card ID (vcd_{id}), Credential ID (c_{id}), Verification Card ID (vc_{id}), Verification Card Set ID

(vcs_{id}), encryption voting options, encrypted partial Choice Return Codes, Control Components computations (those sent in step 2) and 7) of Protocol CreateCC algorithm), list of correctness IDs, verification card public key (K_{id}), verification card public key signature, vote signature, Credential ID signing certificate, Authentication Token, Authentication token signature, cryptographic proofs, ciphertext exponentiations), the digitally Signed Receipt and the signed authentication token in the Ballot Box.

In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter is instructed to restart the login process.

5.5 Protocol GetCC algorithm

The system allows the voter to log out after sending the vote, and to log back in to see the Choice Return Codes again and confirm their vote. In this case, after authentication, the corresponding vote and Control Components computations are obtained from the Ballot Box and the voting server does the following:

1. Obtain the encrypted pre-Choice Return Codes as it is done in step 3.b.
2. Obtain the decrypted pre-Choice Return Codes as it is done in step 8.c.
3. Repeat the steps defined in section 5.4.4.

Using the computations stored in the ballot box we avoid the Control Components from computing the Choice Return Codes more than once for the same voter.

5.6 Confirm a vote

5.6.1 Protocol Confirm algorithm

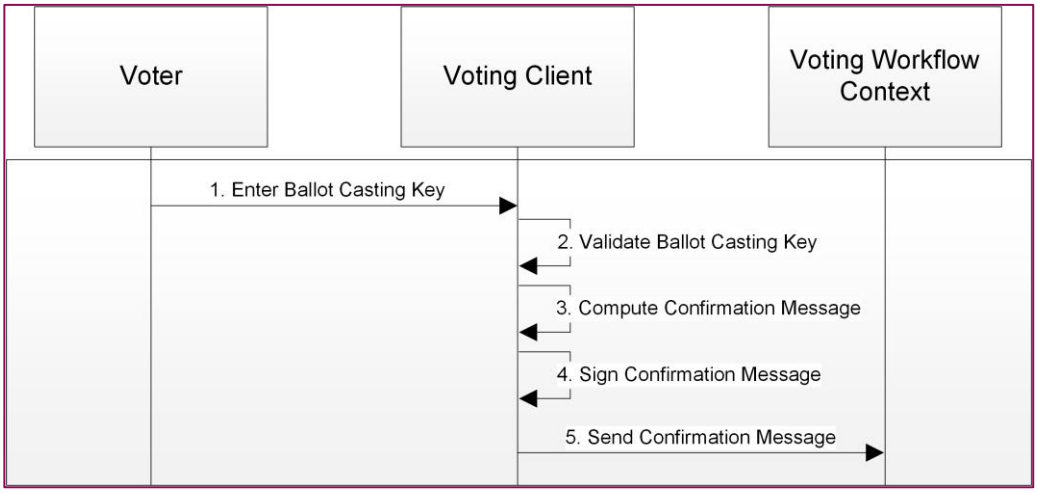

Figure 22 - Protocol Confirm

The following steps are followed to confirm a vote:

1) The voter enters the Ballot Casting Key (BCK^{id}).

2) The **Client Context** uses the checksum algorithm to validate the Ballot Casting Key (BCK^{id}).

3) If the validation is successful, the **Client Context** squares the Ballot Casting Key (BCK^{id}) and raises it to the Verification Card private key (k_{id}): $(BCK^{id})^{2 \cdot k_{id}} \ mod \ p$ (where p is taken from the encryption parameters). This is the Confirmation Message: $CM^{id} = (BCK^{id})^{2 \cdot k_{id}}$.

4) Calls the Digital signature generation primitive to sign the Confirmation Message together with the Authentication Token signature, the Voting Card ID (vcd_{id}) and the Election Event ID ($eeid$) using the Credential ID signing private key ($k^{S}_{C_{id}}$).

5) The exponentiated Confirmation Message, its signature, the Credential ID (c_{id}), the Credential ID signing certificate and the Authentication Token are sent to the **Voting Workflow Context Context.**

5.6.2 Protocol ProcessConfirm algorithm

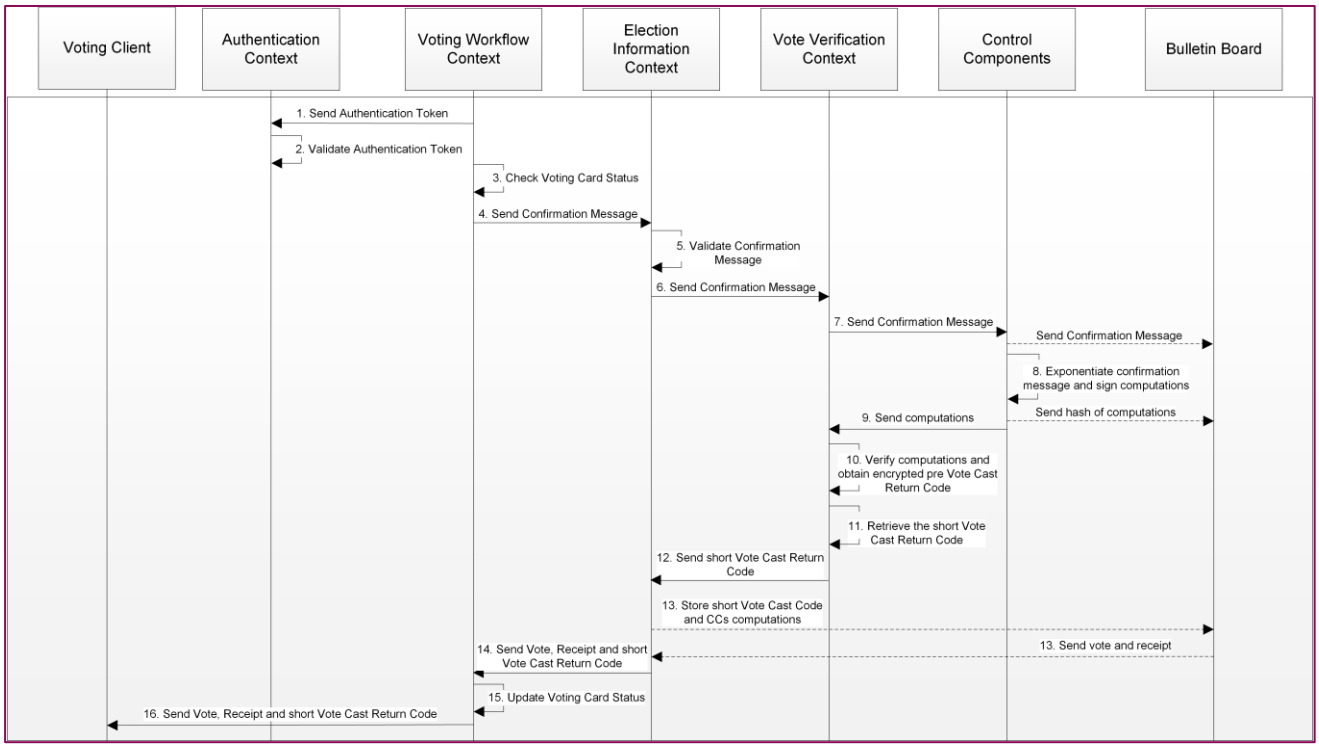

Figure 23 - Protocol ProcessConfirm

Upon receipt of a confirmation message from a voter, the server-side takes the following steps:

1) The **Voting Workflow Context** sends the Authentication Token to the **Authentication Context** to be validated.

2) The **Authentication Context**:

   a) Validates the signature of the Authentication Token, using the Authentication Token Signer certificate.

b) Checks that the fields Tenant ID, Election Event ID ($eeid$), Voting Card ID (vcd_{id}), in the Authentication Token are consistent with the Voting Card ID (vcd_{id}) and Election Event ID ($eeid$) provided.

c) Verifies that the Authentication Token has not expired.

3) If the validation of the Authentication Token is successful, the **Voting Workflow Context** checks the status of the Voting Card.

4) Only when the Voting Card status is SENT BUT NOT CAST, the confirmation message is sent to the Election Information Context to be validated.

5) The **Election Information Context** validates the confirmation message in the following way:

a) Validates that the Ballot Box is not blocked.

b) Checks that the Voting Card ID (vcd_{id}) has not confirmed a vote yet and that exists a vote pending to confirm for that Voting Card ID (vcd_{id}).

c) Checks that the Credential ID signing certificate (which is provided together with the Confirmation Message) is issued for the same Credential ID (c_{id}) which is provided as a parameter.

d) Validates the certificate chain: [Credential ID signing Certificate, Credentials CA, Election Event Root CA].

e) Verifies the digital signature over the Confirmation Message, the Authentication Token signature, the Voting Card ID (vcd_{id}) and the Election Event ID ($eeid$) using the Credential ID signing certificate.

f) Checks that the Election Event ID ($eeid$) signed with the Confirmation Message matches with the Election Event ID ($eeid$) received from the request.

g) Verifies that the Voting Card ID (vcd_{id}) signed with the Confirmation Message corresponds with the Voting Card ID (vcd_{id}) provided in the request.

h) Checks that the Confirmation Message is a group element.

i) Verifies if the election is out of period checking the current timestamp against the election dates from the election event configuration.

In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter can log out and log in, and the steps of GetID, Authentication and GetCC are executed. After that, the voter is presented with the confirmation screen and they can restart the confirmation process.

6) If valid, it is checked whether the maximum number of confirmation attempts has been reached. If so, the protocol stops, an error is returned to the client context, which shows it to the voter.

From this point, the voter is blocked and cannot proceed to confirm their vote. If not, the counter is increased, and the confirmation message is sent to the **Vote Verification Context.**

7) The **Vote Verification Context** then broadcasts the information to every Control Component (CCR_j).

8) Each Control Component (CCR_j):

   a) Checks if the choice return codes have been previously computed for that. If this is not the case, the Control Component stops the process, logs an error and forwards it to the client context, which shows an error message to the voter. In this way we prevent the Control Components from processing a confirmation of a vote that has not been cast.

   b) Checks if the maximum number of confirmation attempts has been reached. If so, the protocol stops, an error is logged and returned to the client context, which shows it to the voter. If not, the counter is increased and the process continues.

   c) Logs the confirmation message and its signature in a format that the signature could be validated by an external process using the information logged.

   d) Derives the Voter Vote Cast Return Code generation private key (kc_{id}^j) from the corresponding CCR_j choice return codes generation private key (k_j') , the confirm text padding and the Verification Card ID (vc_{id}):

      - Computes $kc_{id}^j = KDF\left(vc_{id}||confirm||k_j',\ p\ length\right)$ calling the Key Derivation Function: KDF1 specification primitive.
      - Truncates the result to have 2047 bits.
      - Checks that $1 \leq kc_{id}^j \leq q - 1$
      - If the derived value is equal or greater than q, it computes again a derivation but using as input the derived value $KDF\left(kc_{id}^j,\ p\ length\right)$
      - kc_{id}^j is the Voter Vote Cast Return Code generation private key.

   e) Computes the squared hash and then the exponentiation of the result:

$$(Hash(CM^{id})^2)^{kc_{id}^j}$$

   and calls the Exponentiation proof generation primitive to compute a proof of knowledge of the exponent kc_{id}^j. The inputs of the primitive are the following:

      - Base elements (group elements): $[g, Hash(CM^{id})^2]$
      - Exponents: $[kc_{id}^j]$
      - Public input (group elements): $[Kc_{id}^j, (Hash(CM^{id})^2)^{kc_{id}^j}]$

- Additional information: "$ExponentiationProof$"
- Mathematical group (p, q, g)

The result is $\sigma^3_{CCR_j}$.

f) The exponentiation and the proof $\sigma^3_{CCR_j}$ are signed using the CCR_j signing private key $(sk^s_{CCR_j})$.

9) The signed information

$$\{((Hash(CM^{id})^2)^{kc^1_{id}}, \sigma^3_{CCR_1}); ((Hash(CM^{id})^2)^{kc^2_{id}}, \sigma^3_{CCR_2});$$

$$((Hash(CM^{id})^2)^{kc^3_{id}}, \sigma^3_{CCR_3}); ((Hash(CM^{id})^2)^{kc^4_{id}}, \sigma^3_{CCR_4})\}$$

is sent to the Vote Verification Context.

10) The **Vote Verification Context:**

a) Retrieves the signed Voter vote cast return generation public keys $(Kc^1_{id}, Kc^2_{id}, Kc^3_{id}, Kc^4_{id})$ in the Verification Card Set Control Components Data associated to the voter vc_{id}.

b) Verifies the signatures and the proofs, multiplies the exponentiated squared hashes of the Confirmation Message and obtains the pre-Vote Cast Return Code.

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

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

11) Given the pre-Vote Cast Return Code, the **Vote Verification Context** does the following actions:

a) Looks for the Codes Mapping Table corresponding to the Verification Card ID (vc_{id}) and Election Event ID $(eeid)$.

b) Calls the Message Authentication Code generation primitive the concatenation of the pre-Vote Cast Return Code $(pVCC^{id})$ with the Verification Card ID (vc_{id}) and the Election Event ID $(eeid)$. The result is the long Vote Cast Return Code:

$$lVCC^{id} = Hash(pVCC^{id}||vc_{id}||EEID)$$

c) Computes the hash of the concatenation of the long Vote Cast Return Code $(lVCC^{id})$ and the Codes Secret Key (C_{sk}). Then, calls the Key Derivation Function: KDF1 specification to generate Vote Cast Return Code encryption symmetric key: $skvcc^{id} = KDF(Hash(lVCC^{id}||C_{sk}), 256\ bits)$.

d) Calls the Hash generation primitive to compute the hash of the long Vote Cast Return Code: $Hash(lVCC^{id})$ and find the corresponding value in the table: $Hash(lVCC^{id})$ – $Enc(VCC^{id}||signedVCC^{id}, skvcc^{id})$.

e) Calls the Symmetric decryption primitive to decrypt the ciphertext $Enc(VCC^{id}||signedVCC^{id}, skvcc^{id})$ using the corresponding symmetric key and obtain the Vote Cast Return Code and its signature: $VCC^{id}, signedVCC^{id}$.

f) Verifies the signature over the retrieved Vote Cast Return Code using the Vote Cast Return Code Signer public key ($VCCs_{pk}$).

In case any of the previous actions/validations fail, the application stops and an error is logged and forwarded to the client context, which shows an error message to the voter. The voter can log out and log in, and the steps of GetID, Authentication and GetCC are executed. After that, the voter is presented with the confirmation screen and they can restart the confirmation process.

12) The short Vote Cast Return Code together with its signatures and the Control Components computations are sent to the **Election Information Context.**

13) The **Election Information Context** stores the Vote Cast Return Code, its signature and the Control Component computations in the Ballot Box and retrieves the vote and the receipt.

14) The vote and the receipt are sent to the **Voting Workflow Context**.

15) The **Voting Workflow Context** changes the status of the voting card from “SENT BUT NOT CAST” to “CAST”.

16) The **Voting Workflow Context** sends the receipt, the vote and the Vote Cast Return Code to the Client Context.

5.7 Client-side receipt validation

The **Client Context** validates the received information and shows the Vote Cast Return Code to the voter: The **Client Context**:

1) Verifies the signature of the Signed Receipt, using the Ballot Box Signer public key ($BBs_{pk}^{bb_{id}}$)

2) Calls the Hash generation primitive to compute the hash of the Vote Signature, the Authentication Token signature, the Verification Card public key signature, the Election Event ID ($eeid$) and the Voting Card ID (vcd_{id}). It verifies that this value corresponds with the receipt value received.

3) Verifies that the vote received is the same as the vote sent, if it is still present in memory. Otherwise, it verifies that:

a) The signature of the vote received validates successfully using the Credential ID signing public key ($K^{s}_{C_{id}}$);

b) The signature of the Authentication Token from the vote received validates successfully using the Authentication Token Signer public key (ATs_{pk});

c) The Election Event ID ($eeid$) and the Voting Card ID (vcd_{id}) in the Authentication Token from the vote received match those from the Authentication Token received from the authentication process.

d) Verifies that the Voting Card ID (vcd_{id}) signed with the Vote Cast Return Code matches the one in the received vote.

e) Verifies the signature of the Vote Cast Return Code using the Vote Cast Return Code Signer public key ($VCCs_{pk}$).

If all the verifications succeed, the Vote Cast Return Code is shown to the voter on the screen, and optionally the receipt and signature (in case of the Canton of Neuchâtel, the receipt is shown on the screen, in case of Fribourg, the receipt is not shown). Otherwise, the process stops and an error is shown on the screen.

5.8 Request Vote Cast Return Code and Receipt

The system allows the voter to log out after casting the vote, and then log in back to see the Vote Cast Return Code and the Receipt again. In this case, after the steps defined in the protocol GetID and in the Authentication phase are done, the corresponding Vote Cast Return Code and Receipt are obtained from the Ballot Box and the steps defined in the section 5.7 are repeated.

6 Counting phase

6.1 Protocol Tally algorithm

The following diagram is an overview of the *Tally* algorithm. The modules involved in this protocol are the Election Information Context, the Mixing Control Components ($CCM_1, CCM_2, CCM_3, CCM_4$), the Election Administration and the Bulletin Board.

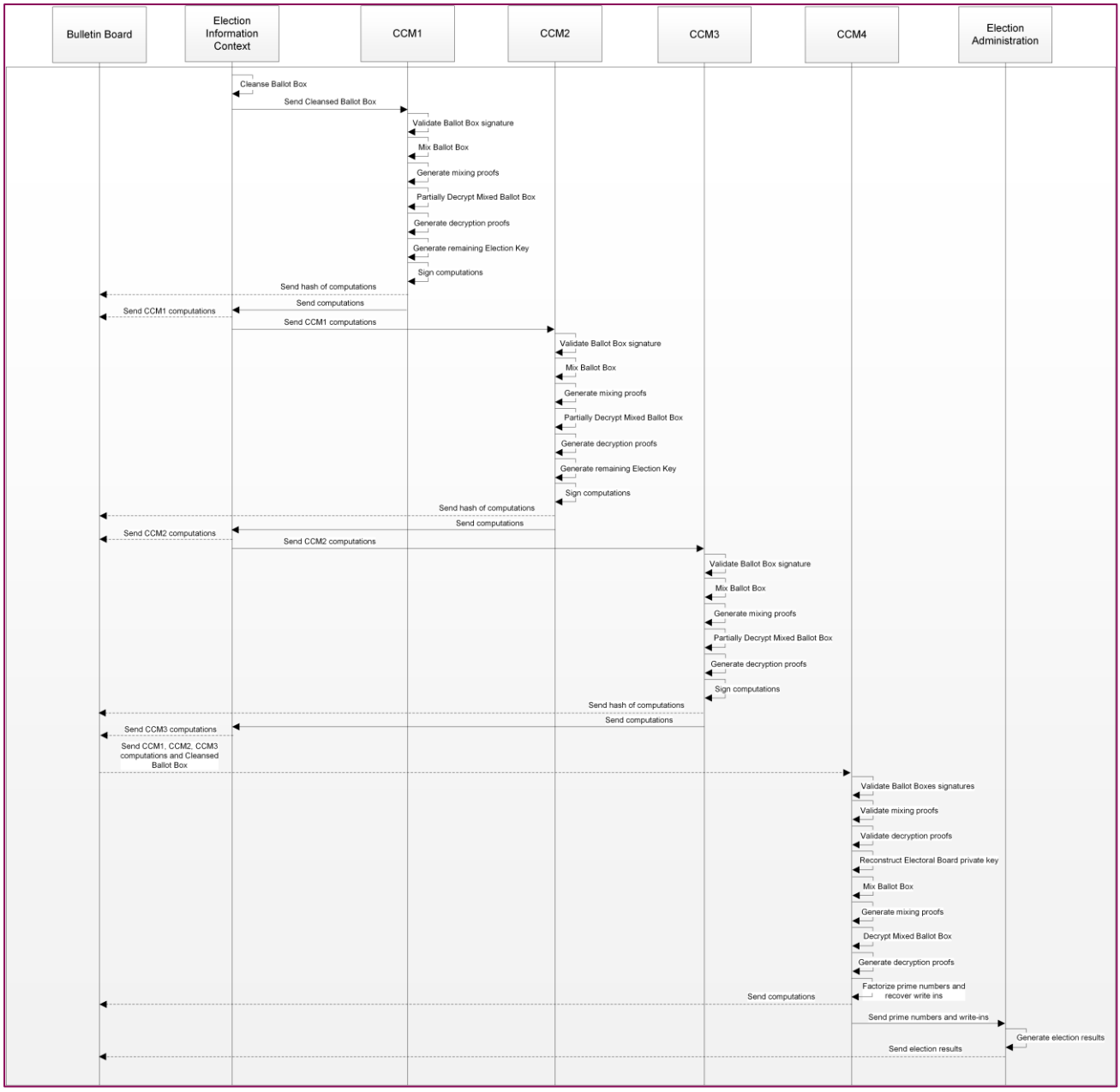

Figure 24 - Counting phase overview

As explained in section 0, each voting system component has a local Bulletin Board where the audit data is stored, and the information of these local Bulletin Boards is then compiled to a global Bulletin

Board (Audit System component). For simplicity, we are not going to refer to each local Bulletin Board but to a global one.

6.1.1 Cleansing

The Cleansing process makes sure that only confirmed votes (i.e.: those for which the voters entered a valid Ballot Casting Key (BCK^{id})) are considered during the mixing and decryption processes. In addition, it ensures that in case there is more than one vote per Voting Card ID (vcd_{id}), none of them are considered in further phases. Actually, this validation is already done by the server during the voting phase, so it is not necessary in this step and for this reason the Cleansing only discards unconfirmed votes. Then, it removes all the information from the valid votes except for the encrypted voting options, which will be processed by the mixing and the decryption.

As the first part of the mixing and decryption processes is done in the online Control Components, the Cleansing is executed online in the Voting Server. The following lists are generated:

- List of non-confirmed votes: list of Voting Card IDs, timestamps and receipts that correspond to non-confirmed votes.
- List of successful votes: list of Voting Card IDs, timestamps and receipts corresponding to the valid and confirmed votes.
- List of ciphertexts (encrypted options) corresponding to valid votes.

The list containing the encrypted options will be the Cleansed Ballot Box, that is, the input of the mixing process.

When cleansed Ballot Boxes are requested from the first Control Component to be mixed, the first step is to sign them in the server-side, to protect their integrity.

Each Ballot Box has an opening and closing time configured inside the election period. A regular Ballot Box cannot be sealed and exported from the server until the election period ends. However, during the configuration it is possible to create Ballot Boxes for testing purposes, which can be sealed and exported before the Ballot Box closing time has passed.

The Cleansed Ballot Box is signed together with a timestamp, the Election key, the Tenant ID, the Election Event ID ($eeid$) and the Ballot Box ID ($bbid$), using the Election Information Signing private key (EI^{s}_{sk}). Both the list of successful votes and the list of failed votes are signed during the Ballot Box export process (see section 6.2).

The whole Ballot Box must be validated in the verification phase, that is, it should be checked that only non-confirmed votes were discarded during the cleansing and that the confirmed ones were valid (i.e.: valid vote signature, valid zero knowledge proofs, …).

6.1.2 Mixing and Decryption

Control Components will perform mixing and partial decryption sequentially.

The mixing process breaks the correlation between the votes collected in the Ballot Boxes and the votes to be decrypted, by shuffling and re-encrypting them. It also produces proofs of the correct computation of the mixing process. The decryption of the votes requires a collaboration between the online Control Components (CCM_1, CCM_2, CCM_3) and the offline Control Component CCM_4.

The Electoral Board members provide their shares of the Electoral Board private key (EB_{sk}) to the CCM_4, where the private key reconstructed using the Shamir Threshold Secret Sharing reconstruction algorithm. At that point, the votes in the Ballot Boxes are decrypted and the result factorized (according to the voting options they could be composed of) to obtain the individual voting option values representing the voter's selections.

Similar to the mixing process, the decryption process also produces zero knowledge proofs of correct computation.

For the description to be clearer, we will differentiate between the first, the second, the third and the last Control Component.

6.1.2.1 First Control Component (CCM_1)

The first Control Component CCM_1 receives as input the following parameters:

- Cleansed Ballot Box.
- Signature of Cleansed Ballot Box, Election key, Tenant ID, Election Event ID ($eeid$), Ballot Box ID ($bbid$) and timestamp.
- Election Information Signing Certificate.
- Election public key (EL_{pk})

And does the following actions:

1) Validate the signature of the Cleansed Ballot Box using the Election Information Signing public key (EI^s_{pk}).
2) Validate the Certificate chain [Election Information Signing Certificate, Platform Root CA].
3) Mix the Cleansed Ballot Box:
    a) Calls the ElGamal ciphertexts permutation primitive with input the Cleansed Ballot Box. The result is the permuted Cleansed Ballot Box.
    b) For each ciphertext in the permuted Cleansed Ballot Box, call the ElGamal Re-encryption primitive with input the ciphertext and the Election key (inside the corresponding Ballot Box Voter Data). The result is the permuted and re-encrypted Cleansed Ballot Box.
4) Create a Mixed Ballot Box CCM_1 with the Ballot Box ID ($bbid$) and the set of permuted and re-encrypted votes.

5) Call the Mixing proof generation primitive to generate the cryptographic proofs to demonstrate that the shuffled and re-encrypted votes in the Mixed Ballot Box CCM_1 are the same that those in the input Ballot Box. Put them in a file together with the corresponding Ballot Box ID $(bbid)$.

6) For each ciphertext in the Mixed Ballot Box CCM_1, call the ElGamal decryption primitive with input the ciphertext and the Control Component CCM_1 Mixing private key (x_1).

7) Create a Partially Decrypted Ballot Box CCM_1 with the Ballot Box ID $(bbid)$ and the set of partially decrypted votes.

8) For each ciphertext in the Partially Decrypted Ballot Box CCM_1, call the Decryption proof generation primitive with the following inputs:

    a) Control Component CCM_1 Mixing public key (g^{x_1}).

    b) Ciphertext.

    c) Partially decrypted ciphertext.

    d) Control Component CCM_1 Mixing private key (x_1).

    e) Mathematical group.

    The output is the decryption proofs π_{dec_1}.

9) When votes are partially decrypted, the contribution of the CCM_1 Mixing private key (x_1) in the Election public key is removed. This means that the next Control Component performs the mixing using the remaining Election public key and this key is computed in the CCM_1:

$$Election\ public\ key\text{: } EL_{pk} = (EL_{pk_1}, EL_{pk_2}, \dots, EL_{pk_m}) = (g^{EB_{sk}+\sum_{j=1}^{3} x_j^1}, g^{EB_{sk}+\sum_{j=1}^{3} x_j^2}, \dots, g^{EB_{sk}+\sum_{j=1}^{3} x_j^m})$$

$Remaining\ Election\ public\ key$:

$$EL_{pk}^1 = \left(g^{EB_{sk}+\sum_{j=1}^{3} x_j^1} \cdot g^{-x_1^1}, g^{EB_{sk}+\sum_{j=1}^{3} x_j^2} \cdot g^{-x_1^2}, \dots, g^{EB_{sk}+\sum_{j=1}^{3} x_j^m} \cdot g^{-x_1^m}\right)$$
$$= \left(g^{EB_{sk}+x_2^1+x_3^1}, g^{EB_{sk}+x_2^2+x_3^2}, \dots, g^{EB_{sk}+x_2^m+x_3^m}\right)$$

10) Create a timestamp, and call the Digital signature generation primitive to sign the timestamp together with the Cleansed Ballot Box, Mixed Ballot Box CCM_1, the Partially Decrypted Ballot Box CCM_1, the Mixing proof, the Decryption proofs, the Remaining Election Key, the Election key, the Election Event ID $(eeid)$, the Ballot Box ID $(bbid)$ and an identifier of the Control Component, using the CCM_1 signing private key $(sk_{CCM_1}^s)$.

11) The following information is sent to the next Control Component:

    a) Mixed Ballot Box CCM_1 and mixing proof.

    b) Partially Decrypted Ballot Box CCM_1 and decryption proofs.

    c) Remaining Election key EL_{pk}^1

d) CCM_1 Mixing public key (g^{x_1}).

e) CCM_1 signing certificate and CCM_1 CA certificate.

f) Signature of the timestamp, Cleansed Ballot Box, Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixing proof, Decryption proofs, Remaining Election Key, Election key Election Event ID ($eeid$), Ballot Box ID ($bbid$) and an identifier of the Control Component CCM_1.

6.1.2.2 Second Control Component (CCM_2)

The second Control Component CCM_2 receives as input the following parameters:

- Cleansed Ballot Box, Mixed Ballot Box CCM_1 and Partially Decrypted Ballot Box CCM_1.
- Mixing proof and Decryption proofs
- Signature of the timestamp, Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixing proof, Decryption proofs, Remaining Election Key, Election Event ID ($eeid$), Ballot Box ID ($bbid$) and an identifier of the Control Component CCM_1.
- CCM_1 signing certificate, CCM_1 CA certificate and Platform Root CA.
- CCM_1 Mixing public key (g^{x_1}).
- Remaining Election public Key: EL_{pk}^1
- Election public key: EL_{pk}.

and does the following actions:

1) Validate the signature of the output of the previous Control Components using the CCM_1 signing public key ($pk_{CCM_1}^S$).
2) Validate the certificate chain [CCM_1 signing certificate, CCM_1 CA, and Platform Root CA].
3) Mix the Partially Decrypted Ballot Box CCM_1:
    a) Call the ElGamal ciphertexts permutation primitive with input the Partially Decrypted Ballot Box CCM_1. The result is the permuted Partially Decrypted Ballot Box CCM_1.
    b) For each ciphertext in the permuted Partially Decrypted Ballot Box CCM_1, call the ElGamal Re-encryption primitive with input the ciphertext and the remaining Election Key: EL_{pk}^1. The result is the permuted and re-encrypted Partially Decrypted Ballot Box CCM_1.
4) Create a Mixed Ballot Box CCM_2 with the Ballot Box ID ($bbid$) and the set of shuffled and re-encrypted votes.

5) Call the Mixing proof generation primitive to generate the cryptographic proof to demonstrate that the shuffled and re-encrypted votes in the Mixed Ballot Box CCM_2 are the same that those in the input Ballot Box. Put them in a file together with the corresponding Ballot Box ID $(bbid)$.

6) For each ciphertext in the Mixed Ballot Box CCM_2, call the ElGamal decryption primitive with input the ciphertext and the Control Component CCM_2 Mixing private key (x_2).

7) Create a Partially Decrypted Ballot Box CCM_2 with the Ballot Box ID $(bbid)$ and the set of partially decrypted votes.

8) For each ciphertext in the Partially Decrypted Ballot Box CCM_2, call the Decryption proof generation primitive with the following inputs:

    a) Control Component CCM_2 Mixing public key (g^{x_2}).

    b) Ciphertext.

    c) Partially decrypted ciphertext.

    d) Control Component CCM_2 Mixing private key (x_2).

    e) Mathematical group.

    The output is the decryption proofs π_{dec_2}.

9) When votes are partially decrypted, the contribution of the CCM_2 Mixing private key (x_2) in the Election key is removed. This means that the next Control Component performs the mixing using the remaining Election key and this key is computed in the CCM_2:

$$Actual\ Election\ public\ key\text{:}\ EL_{pk}^{1} = (g^{EB_{sk}+\sum_{j=1}^{3} x_j^1} \cdot g^{-x_1^1}, g^{EB_{sk}+\sum_{j=1}^{3} x_j^2} \cdot g^{-x_1^2}, \dots, g^{EB_{sk}+\sum_{j=1}^{3} x_j^m} \cdot g^{-x_1^m})$$

$$Remaining\ Election\ public\ key\text{:}\ EL_{pk}^{2} = \left(g^{EB_{sk}+x_2^1+x_3^1} \cdot g^{-x_2^1}, g^{EB_{sk}+x_2^2+x_3^2} \cdot g^{-x_2^2}, \dots, g^{EB_{sk}+x_2^m+x_3^m} \cdot g^{-x_2^m}\right)$$
$$= (g^{EB_{sk}+x_3^1}, g^{EB_{sk}+x_3^2}, \dots, g^{EB_{sk}+x_3^m})$$

10) Create a timestamp, and call the Digital signature generation primitive to sign the timestamp together with the Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, the Partially Decrypted Ballot Box CCM_2, the Mixing proof, the Decryption proofs, the Remaining Election Key, the Election Event ID $(eeid)$, the Ballot Box ID $(bbid)$ and an identifier of the Control Component, using the CCM_2 signing private key $(sk_{CCR_j}^{s})$.

11) The following information is sent to the next Control Component:

    a) Mixed Ballot Box CCM_2 and mixing proof.

    b) Partially Decrypted Ballot Box CCM_2 and decryption proofs.

    c) Remaining Election public keys: EL_{pk}^{1}, EL_{pk}^{2}

    d) CCM_2 Mixing public key (g^{x_2}).

e) CCM_2 signing certificate and CCM_2 CA certificate.

f) Signature of the timestamp, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixing proof, Decryption proofs, Remaining Election Key, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_2.

6.1.2.3 Third Control Component (CCM_3)

The third Control Component CCM_3 receives as input the following parameters:

- Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2 and Partially Decrypted Ballot Box CCM_2.
- Mixing proof and Decryption proofs
- Signature of the timestamp, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixing proof, Decryption proofs, Remaining Election Key, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_2.
- CCM_2 signing certificate, CCM_2 CA certificate and Platform Root CA.
- CCM_2 Mixing public key (g^{x_2}).
- Remaining Election public Key: EL_{pk}^1, EL_{pk}^2

and does the following actions:

1) Validate the signature of the output of the previous Control Components using the CCM_2 signing certificate.
2) Validate Certificate chain [CCM_2 election signing, CCM_2 CA, and Platform Root CA].
3) Mix the Partially Decrypted Ballot Box CCM_2:
    a) Call the ElGamal ciphertexts permutation primitive with input the Partially Decrypted Ballot Box CCM_2. The result is the permuted Partially Decrypted Ballot Box CCM_2.
    b) For each ciphertext in the permuted Partially Decrypted Ballot Box CCM_2, call the ElGamal Re-encryption primitive with input the ciphertext and the remaining Election Key: EL_{pk}^2. The result is the permuted and re-encrypted Partially Decrypted Ballot Box CCM_2.
4) Create a Mixed Ballot Box CCM_3 with the Ballot Box ID $(bbid)$ and the set of shuffled and re-encrypted votes.
5) Call the Mixing proof generation primitive to generate the cryptographic proof to demonstrate that the shuffled and re-encrypted votes in the Mixed Ballot Box CCM_3 are the same that those in the input Ballot Box. Put them in a file together with the corresponding Ballot Box ID $(bbid)$.

6) For each ciphertext in the Mixed Ballot Box CCM_3, call the ElGamal decryption primitive with input the ciphertext and the Control Component CCM_3 Mixing private key (x_3).

7) Create a Partially Decrypted Ballot Box CCM_3 with the Ballot Box ID $(bbid)$ and the set of partially decrypted votes.

8) For each ciphertext in the Partially Decrypted Ballot Box CCM_3, call the Decryption proof generation primitive with the following inputs:

    a) Control Component CCM_3 Mixing public key (g^{x_3}).

    b) Ciphertext.

    c) Partially decrypted ciphertext.

    d) Control Component CCM_3 Mixing private key (x_3).

    e) Mathematical group.

    The output is the decryption proofs π_{dec_3}.

9) Notice that when votes are partially decrypted the contribution of the CCM_3 Mixing private key (x_3).in the Election key is removed. In this case, the remaining Election key will be directly the Electoral board public key (EB_{pk}). and it is not necessary to compute it.

10) Create a timestamp, and call the Digital signature generation primitive to sign the timestamp together with the Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, the Partially Decrypted Ballot Box CCM_3, the Mixing proof, the Decryption proofs, the Remaining Election Key, the Election Event ID $(eeid)$, the Ballot Box ID $(bbid)$ and an identifier of the Control Component, using the CCM_3 signing private key $(sk^S_{CCM_3})$.

11) The following information is sent to the next Control Component:

    a) Mixed Ballot Box CCM_3 and mixing proof

    b) Partially Decrypted Ballot Box CCM_3 and decryption proofs

    c) CCM_3 Mixing public key (g^{x_3})

    d) CCM_3 signing certificate and CCM_3 CA certificate

    e) Signature of the timestamp, Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, Partially Decrypted Ballot Box CCM_3, Mixing proof, Decryption proofs, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_3.

6.1.2.4 AuditorVerify algorithm

Before performing the last mixing and decryption in the last Control Component CCM_4, blocks 1, 2 and 3 from the Verifier tool [3] must be run (except the ones requiring the data provided by the offline control component) in order to ensure the privacy of the process.

This tool will receive as input the following information:

- Ballot Box, Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, Partially Decrypted Ballot Box CCM_3
- CCM_1, CCM_2 and CCM_3 Mixing and Decryption proofs
- CCM_1, CCM_2 and CCM_3 Mixing public keys $(g^{x_1}, g^{x_2}, g^{x_3})$.
- Election public key (EL_{pk}).
- Remaining Election public Keys: EL_{pk}^1, EL_{pk}^2
- CCM_1, CCM_2 and CCM_3 Signing Certificates
- CCM_1, CCM_2 and CCM_3 CA Certificates
- Platform Root CA
- Election Information Signing Certificate
- Signature of the timestamp, Cleansed Ballot Box , Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixing proof, Decryption proofs, Remaining Election public Key, Election public key, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_1Signature of the timestamp, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixing proof, Decryption proofs, Remaining Election public key, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_2.
- Signature of the timestamp, Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, Partially Decrypted Ballot Box CCM_3, Mixing proof, Decryption proofs, Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and an identifier of the Control Component CCM_3.

and will perform the following validations:

- Validate Certificate chains [CCM_j signing certificate, CCM_j CA, and Platform Root CA].
- Validate Certificate chain [Election Information Signing Certificate, Platform Root CA]
- Validate the signature of the output of the previous Control Components using the corresponding CCM_j signing certificate.
- Validate the signature of the Cleansed Ballot Box using the Election Information Signing Certificate.
- Validate CCM_1, CCM_2 and CCM_3 Mixing proofs.
- Validate CCM_1, CCM_2 and CCM_3 Decryption proofs.

In case all the validations are successful, the process continues and the mixing and decryption in the last node are executed. Nevertheless, if some of the validations fail, the process is stopped since either the Voting Server or one of the Control Components have misbehaved.

6.1.2.5 Offline Control Component (CCM_4)

The last Control Component CCM_4 receives as input the output of the previous Control Components:

- Cleansed Ballot Box, Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, Partially Decrypted Ballot Box CCM_3.
- CCM_1, CCM_2 and CCM_3 Mixing and Decryption proofs.
- CCM_1, CCM_2 and CCM_3 Mixing public keys ($g^{x_1}, g^{x_2}, g^{x_3}$).
- Election public key (EL_{pk}).
- Remaining Election public keys: EL_{pk}^1, EL_{pk}^2
- CCM_1, CCM_2 and CCM_3 Signing Certificates.
- CCM_1, CCM_2 and CCM_3 CA Certificates.
- Platform Root CA.
- Election Information Signing Certificate.
- Signature of the timestamp, Cleansed Ballot Box , Mixed Ballot Box CCM_1, Partially Decrypted Ballot Box CCM_1, Mixing proof, Decryption proofs, Remaining Election public key, Election public key, Election Event ID ($eeid$), Ballot Box ID ($bbid$) and an identifier of the Control Component CCM_1Signature of the timestamp, Partially Decrypted Ballot Box CCM_1, Mixed Ballot Box CCM_2, Partially Decrypted Ballot Box CCM_2, Mixing proof, Decryption proofs, Remaining Election public key, Election Event ID ($eeid$), Ballot Box ID ($bbid$) and an identifier of the Control Component CCM_2.
- Signature of the timestamp, Partially Decrypted Ballot Box CCM_2, Mixed Ballot Box CCM_3, Partially Decrypted Ballot Box CCM_3, Mixing proof, Decryption proofs, Election Event ID ($eeid$), Ballot Box ID ($bbid$) and an identifier of the Control Component CCM_3.

And does the following actions:

1) The Electoral Board private key (EB_{sk}) is reconstructed from the shares. If the Electoral Board private key has multiple components, all of them are reconstructed.
2) Validate the signature of the output of the previous Control Component using the corresponding CCM_j signing public key ($pk_{CCM_j}^s$).
3) Validate Certificate chains [CCM_j signing certificate, CCM_j CA, and Platform Root CA].

4) Mix the Partially Decrypted Ballot Box CCM_3:

    a) Call the ElGamal ciphertexts permutation primitive with input the Partially Decrypted Ballot Box CCM_3. The result is the permuted Partially Decrypted Ballot Box CCM_3.

    b) For each ciphertext in the permuted Partially Decrypted Ballot Box CCM_3, call the ElGamal Re-encryption primitive with input the ciphertext and the Electoral Board public key (EB_{pk}). The result is the permuted and re-encrypted Partially Decrypted Ballot Box CCM_3.

5) Create a Mixed Ballot Box CCM_4 with the Ballot Box ID $(bbid)$ and the set of shuffled and re-encrypted votes.

6) Call the Mixing proof generation primitive to generate the cryptographic proof to demonstrate that the shuffled and re-encrypted votes in the Mixed Ballot Box CCM_4 are the same that those in the Partially Decrypted Ballot Box CCM_3. Put them in a file together with the corresponding Ballot Box ID $(bbid)$.

7) For each ciphertext in each Mixed Ballot Box CCM_4:

    a) Call the ElGamal decryption primitive with input the ciphertext and the reconstructed Electoral Board private key (EB_{sk}). The result is a product of primes representing a set of voting options and if write-ins are allowed by the election, the process will also recover as many elements as write-ins are specified in the ballot. The result is a product of primes representing a set of voting options and if write-ins are allowed by the election, the process will also recover as many elements as write-ins are specified in the ballot.

    b) Generate a decryption proof calling the Decryption proof generation primitive with the following inputs:

        i. Electoral Board public key (EB_{pk}).

        ii. Ciphertext.

        iii. Plaintext.

        iv. Electoral Board private key (EB_{sk}).

        v. Mathematical group.

    c) The product of primes is factorized according to the vote option values and the election rules present in the ballot indicated by the Ballot Box Voter Data. The voting options are recovered, and the following vote correctness validations are done using the *decryptedCorrectnessRule* defined in the ballot:

        i. The minimum selected options, excluding blank selections, is equal to, or greater than, the minimum established (except if allow blank vote is set to true and the whole vote is blank).

ii. The maximum selected options, including blank selections, is equal to, or less than, the maximum established.

iii. The number of times that a candidate has been selected, including candidates that belong to more than one list, is not greater than the contest's accumulation value.

iv. There is no option representation (different from real candidates) that has been selected more than once.

These correctness validations are already performed during the voting phase and therefore, a ballot that has been successfully stored in the ballot box, cleansed, mixed and decrypted is always supposed to pass these checks.

d) If write-ins are present in the decrypted vote, the decrypted values have to be converted back to text:

i. Compute $\sqrt{numeric\ message}\ mod\ p$

ii. Convert the result to UTF-8 representation.

The write-in texts are recovered, and the following vote correctness validations are done:

i. The text used for write-ins does not include the special character #

ii. The representation of the write-in is included (by the system, not by the voter) next to the written-in text, separated by #.

iii. In case the voter has used the write-in for voting for an option, the recovered write-in will be a text. Otherwise, if the write-in has not been used, the recovered value will be the number 2. This is done because the ciphertext corresponding to the encrypted partial choice return codes must have the same number of elements for every voter whether they have filled in the write-ins or not.

8) Generate the following lists for each Mixed Ballot Box, each one containing the Ballot Box ID $(bbid)$:

a) List of encrypted votes, decrypted votes and proofs of correct decryption.

b) List of decrypted votes for which factorization errors happened and result of the factorization (up to the point they were factorized). Usually these errors should not happen since the value encrypted by the voting client is the product of valid prime numbers included in the ballot.

c) List of decrypted votes for which converting error happened, and the result of the decryption (If write-ins are present).

d) List of individual voting option values per each decrypted vote (for the votes that could be factorized) and the write-in text messages (if write-ins are present in the decrypted vote).

9) The Administration Board private key (AB_{sk}) is reconstructed from the shares.

10) For each of the lists, a timestamp is generated, and it is signed together with the list and the Ballot Box ID $(bbid)$ and an identifier of the Decryption process, using the Administration Board private key (AB_{sk}).

6.2 Ballot Box export

After the mixing and decryption is done in CCM_1, CCM_2 and CCM_3 the outputs are downloaded to the last node to perform the final mixing and decryption, as it is explained in the previous section. Additionally, the output of the cleansing and the whole ballot box are signed using the Ballot Box Signer private key ($BBs_{sk}^{bb_{id}}$) and downloaded.

- Generate a timestamp taking the current time as value. Call the Digital signature generation primitive to sign the list of successful votes, the Closing Timestamp, the Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and Tenant ID using the Ballot Box Signer private key ($BBs_{sk}^{bb_{id}}$)
- Generate a timestamp taking the current time as value. Call the Digital signature generation primitive to sign the list of non-confirmed votes, the Closing Timestamp, the Election Event ID $(eeid)$, Ballot Box ID $(bbid)$ and Tenant ID using the Ballot Box Signer private key ($BBs_{sk}^{bb_{id}}$)
- Generate a timestamp taking the current time as value. Call the Digital signature generation primitive to sign the whole Ballot Box, the Closing Timestamp, the Election Event ID $(eeid)$, Ballot Box ID $(bbid)$, the Ballot ID (bid) and Tenant ID using the Ballot Box Signer private key ($BBs_{sk}^{bb_{id}}$).

Ballot Box
Contains one row per vote with the following information: - List of IDs: Tenant ID, Election Event ID ($\boldsymbol{eeid}$), Ballot ID ($\boldsymbol{bid}$), Ballot Box ID ($\boldsymbol{bbid}$), Voting Card ID ($\boldsymbol{vcd_{id}}$), Credential ID ($\boldsymbol{c_{id}}$), Verification Card ID ($\boldsymbol{vc_{id}}$), Verification Card Set ID ($\boldsymbol{vcs_{id}}$) - Encrypted vote (encrypted options, encrypted partial choice codes and encrypted write ins) and its signature - Correctness IDs - Verification Card Public Key ($\boldsymbol{K_{id}}$) and its signature - Authentication Token and its signature - Proofs: Schnorr proof ($\boldsymbol{\pi_{sch}}$), Exponentiation proof ($\boldsymbol{\pi_{exp}}$) and Plaintext Equality proof ($\boldsymbol{\pi_{pleqenc}}$) - Ciphertext exponentiations - Credential ID signing certificate - Receipt and its signature - Choice Return Codes Computations - Vote Cast Return Code Computations - Vote Cast Code and its signature

Table 30 - Ballot Box

7 Audit phase (VerifyTally algorithm)

The following steps must be validated by the auditors in order to verify that the election results accurately reflect the intention of legitimate voters:

1) Configuration
    a) Check the encryption parameters
    b) Check the voting options
    c) Check the number of authentication data generated
2) Ballot Box
    a) Verify the vote signature.
    b) Verify the zero-knowledge proofs
        i. Control Components proofs, computed during the exponentiation of the encrypted partial choice return codes, the partial decryption of the encrypted pre-choice return codes and the exponentiation of the confirmation message.
3) Control Components Secure Logs: (check their consistency with the contents of the Ballot Box):
    a) Verify logs integrity and logs authenticity.
    b) Verify that all the votes stored in the Ballot Box have been processed by the Control Components.
    c) Verify that all the votes processed by the Control Components are stored in the Ballot Box.
    d) Verify that all the confirmations stored in the Ballot Box have been processed by the Control Components.
    e) Verify that all the confirmations processed by the Control Components are stored in the Ballot Box.
    f) Verify that the Choice Return Codes have been computed only once per voter.
    g) Verify that the Vote Cast Return Code has been computed, at most, 5 times.
4) Cleansing
    a) Verify that only confirmed votes have been considered.
    b) Verify that no confirmed votes have been removed.
5) Mixing and Decryption
    a) Verify the mixing and decryption proofs to ensure that no votes have been modified, added or removed during the mixing and decryption processes.

b) Verify the signature of each Mixing Control Component output.

6) Vote factorization

a) Verify that the result of the factorization corresponds to valid options in the election.

In order to perform all these validations, an auditor can use the specification given in [4] to implement their own verifications, or use a software called Verifier [3] to check that the election outcome is correct. This Verifier tool groups the verifications in four blocks:

1. **Block 1:** Pre-Election Verification: used to validate step 1).
2. **Block 2**: Ballot Box Verification: used to validate steps 2) and 3).
3. **Block 3:** Mixing Decryption Verification: used to validate steps 4), 5) and 6).
4. **Block 4:** Result Verification: used to validate correctness of the tallying and the consolidated end results. As the results consolidation is not part of the voting protocol, it is not specified in this document.

8 References

[1] S. F. Chancellery, "Federal Chancellery Ordinance on Electronic Voting (VEleS)," 2018.

[2] S. F. Chancellery, "Annex of the Federal Chancellery Ordinance on Electronic Voting," 2018.

[3] Swisspost, "Verifier_Detailed_specifications," 2018.

[4] Scytl, "Scytl sVote Auditability with Control Components_3.0," 2018.

[5] *PKCS #1 v2.2: RSA Cryptography Standard.*

[6] 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.

[7] D. Chaum and T. Pedersen, "Wallet Databases with Observers.," in *Advances in Cryptology - CRYPTO ' 92, 12th Annual International Cryptology Conference*, Santa Barbara, California, USA, 1992.

[8] A. Shamir, "How to Share a Secret," in *Communications of ACM*, 1979.

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

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

[11] *NIST. FIPS PUB 198-1. The Keyed-Hash Message Authentication Code (HMAC),* July: 2008.

[12] *NIST. FIPS PUB 197. Advanced Encryption Standard (AES),* November, 2001.

[13] *NIST. SP 800-38D. Recommendation for Block Cipher Modes of Operation : Galois/Counter Mode (GCM) and GMAC,* November, 2007.

[14] T. Koshiba and K. Kurosawa, "Short Exponent Diffie-Hellman Problems," in *Public Key Cryptography - PKC 2004*, 2004.

[15] Scytl, "PRO_SP_RS_Audit_Protocol_Control_Components_3.1," 2018.

9 Appendix

9.1 Cryptographic primitives

9.1.1 RSA Key pair generation

Input

- Key length: 2048 bits.

Operation

- The RSA key pair is computed according to the standard PKCS#1 v2.2 [5] with two prime factors .

Output

- RSA key pair.

9.1.2 ElGamal Key pair generation

Input

- The mathematical group defined by (p, q, g).
- The max number N of elements composing the key pair.

Operation

- Generate N random exponents $(sk_1, \dots, sk_N)$ smaller than the value of q using the Random value generation primitive. These random exponents are the private key: $sk = (sk_1, \dots, sk_N)$.
- Compute $pk_1 = g^{sk_1}, pk_2 = g^{sk_2}, \dots, pk_N = g^{sk_N}$ all of them modulo p.
- The result is the public key $pk = (pk_1, \dots, pk_N)$.

Output

- The ElGamal key pair (pk, sk).

9.1.3 X509 certificate generation

Input

- CA (issuer) Private Signing Key.
- Public Key.
- Subject: Common Name, Organization, Organizational Unit and Country.
- Issuer: Common Name, Organization, Organizational Unit and Country.
- Key type.
- Validity period.

- Signature algorithm: RSA-PSS, SHA256, Key length: 2048 bits, Provider: SUN/Forge.

Operation

- Compose the issuer of the certificate using the issuer information provided:
    - Common name
    - Organization
    - Organizational unit
    - Country
- Compose the subject of the certificate using the subject information provided:
    - Common name
    - Organization
    - Organizational unit
    - Country
- Create a serial number:
    - Call the Random value generation with input: type=byte, number=20.
- Create certificate extensions:
    - Basic Constraints:
        - In case key type is “CA”, flag CA=true
        - In case key type is not “CA”, flag CA=false
    - Key usage:
        - In case key type is “CA”, key usage= keycertSign, cRLSign.
        - In case key type is “Sign”, key usage= digitalSignature, nonRepudiation.
        - In case key type is “Encryption”, key usage= keyEncipherment, dataEncipherment.
- Create the public key information using:
    - The public key.
    - The algorithm for which its use is intended.
- Compose the to-be-signed (*tbs*) content of the certificate, using:
    - Issuer
    - Subject

- Serial number
- Version number
- Validity period
- Public key information
- Certificate extensions

- Call the Digital signature generation primitive to sign the *tbs* of the certificate using the CA (issuer) private key, according to the defined algorithm and provider.
- Compose the X.509 certificate using the *tbs* content, the signature algorithm, and the signature value.

Output

- X.509 Certificate

9.1.4 Schnorr proof generation

Input

- Base element (group element)
- Exponent
- Public input (group element)
- Additional information
- Mathematical group

Operation

- Call the Maurer's Unified Proofs Prover primitive with the following inputs:
  - Mathematical group
  - The function PHI defined by:
    - Number of inputs = number of exponents = 1
    - Number of outputs = number of elements of the public input array = 1
    - Base element
    - Computation rules $[(1,1)]$
  - Input data:
    - An array of 1 group element = [Public input]
    - An array of 1 exponent = [Exponent]

- Auxiliary string = additional information

- The result is the Schnorr Proof.

Output

- Schnorr proof

9.1.5 Exponentiation proof generation

Input

- Base Elements array of length k
- Exponent
- Public Input array of length k
- Additional information
- Mathematical group

Operation

- Call the Maurer's Unified Proofs Prover primitive with the following inputs:
  - Mathematical group
  - The function PHI defined by:
    - Number of inputs = number of exponents = 1
    - Number of outputs = number of elements of the public input array = k
    - Base elements
    - Computation rules $[(1,1); (2,1); \dots; (k, 1)]$
  - Input data:
    - An array of k group element = [Public input]
    - An array of 1 exponent = [Exponent]
    - Auxiliary string = additional information

- The result is the Exponentiation proof.

Output

- Exponentiation proof

9.1.6 Plaintext Equality proof generation

Input

- Primary Ciphertext containing $k + 1$ elements for some non-fixed k (k must be at least 1): $[C_0, C_1, \dots, C_k]$.

- Primary public key (which consists of k group elements).
- Primary randomness.
- Secondary Ciphertext containing $k + 1$ elements $[D_0, D_1, \dots, D_k]$.
- Secondary public key (which consists of k group elements).
- Secondary randomness.
- Additional information.
- Mathematical group.

Operation

- Consider the last k elements of the Primary Ciphertext and the last k elements of the Secondary Ciphertext. Call them the *Primary SubCiphertext* $[C_1, \dots, C_k]$ and *Secondary SubCiphertext* $[D_1, \dots, D_k]$ respectively.
- For each i from 1 to k compute $C_i \cdot (D_i)^{-1} \; mod \; p$. The result is the *Divided Ciphertext*, which consists of k group elements.
- Compute the inverse of each element of the Secondary public key (the inverse is computed in the mathematical group). Call the result *Secondary Inverted public key*.
- Call the Maurer's Unified Proofs Prover primitive with the following inputs:
    - Mathematical group.
    - The function PHI defined by:
        - Number of inputs = number of exponents = $[PrimaryRandomness, SecondaryRandomness]$
        - Number of outputs = number of elements of the public input array = $[C_0, D_0, C_1 \cdot (D_1)^{-1}, \dots, C_k \cdot (D_k)^{-1}]$
        - Base elements: $[g, PrimaryPublicKey, SecondaryInvertedPublicKey]$
        - Computation rules $[(1,1); (1,2); (2,1), (2 + k, 2); (3,1), (3 + k, 2); \dots; (k + 1,1), (2k + 1,2)]$
    - Input data:
        - An array of $k + 2$ group element = $[C_0, D_0, C_1 \cdot (D_1)^{-1}, \dots, C_k \cdot (D_k)^{-1}]$
        - An array of 2 exponent = $[PrimaryRandomness, SecondaryRandomness]$
        - Auxiliary string = additional information
- The result is the Plaintext Equality Proof.

Output

- Plaintext equality proof

9.1.7 Mixing proof generation

Based on Bayer and Groth proof of a shuffle [6].

Note: 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 (generate it using the Random value generation primitive)
    - 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

$\vec{c}_A = (com_{ck}(\overrightarrow{A_1}; r_1), \ldots, com_{ck}(\overrightarrow{A_m}; r_m))$ and the vector of randomness as $\vec{r} = (r_1, \ldots r_m)$.

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 $\vec{a} = \{a_1, \ldots, a_N\}$, compute the exponentiation of x to each element on $\vec{a}$: $x^{a_i} \; mod \; p$. The result is $\vec{b} = \{b_1, \ldots, b_N\} = \{x^{a_1}, \ldots, x^{a_N}\}$.

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 $i = 1, \ldots, m$) using the Commitment generation primitive with the following inputs:
   - A random exponent $s_i \in \mathbb{Z}_q$ between 1 and q-1 (generate it using the Random value generation primitive)
   - List of elements to be committed: $\overrightarrow{B_i}$
   - Commitment key $ck = (G_1, \ldots, G_n, H)$

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:

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

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

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:

$$t_1 = y \cdot r_1 + s_1$$
$$\vdots$$
$$t_m = y \cdot r_m + s_m$$

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 $\vec{x} = (x^1, \dots, x^N)$.
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 → $\tilde{a}_1, \ldots, \tilde{a}_n$
  - 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

9.1.8 Decryption proof generation

Based in the Chaum-Pedersen protocol for proving equality of discrete logarithms [7].

Input

- Public key containing *k* elements for some non-fixed *k* (*k* must be at least 1): $(pk_1, \ldots, pk_k)$
- Ciphertext containing k+1 elements: $(C_0, C_1, C_2, \ldots, C_k)$.
- Plaintext containing k elements: $(P_1, P_2, \ldots, P_k)$
- Private Key containing k elements: $(sk_1, \ldots, sk_k)$.
- Mathematical group

Operation

- For each element in the ciphertext array, besides the first one, compute C_i/P_i in the mathematical group, for $i = 1, \dots, k$. The result is the array of divided ciphertexts: $(C'_1, C'_2, \dots, C'_k)$.
- Call the Maurer's Unified Proofs Prover primitive with the following inputs:
    1. Mathematical group
    2. Function PHI
        - Number of inputs = number of private key elements $= k$
        - Number of outputs $= 2k$
        - Array of base elements = $[g\ (from\ the\ mathematical\ group), C_0]$
        - Computation rules = $[(1,1); (2,1); (1,2); (2,2); (1,3); (2,3); \dots; (1,k); (2,k)]$
    3. Input data
        - An array $2k$ of group elements = $[pk_1, C'_1, pk_2, C'_2, \dots, pk_k, C'_k]$.
        - An array of exponents = $(sk_1, \dots, sk_k)$
        - Auxiliary string = additional information
- The result is the decryption proof

Output

- Decryption proof

9.1.9 Shamir Threshold Secret Sharing split algorithm

This algorithm follows the specification from [8]:

Input

- A secret s
- A number of shares k
- A reconstruction threshold t

Operation

- Compute a primer number larger than s: this will be modulo r.
- Compute t-1 random positive integers a_i between 1 and r-1, and let a_0 be s.
- Build the polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}$.
- Each share is computed as $(i, f(i))$ for $i = 1, \dots, k$.

Output

- List of shares $(i, f(i))$ for $i = 1, \dots, k$.

9.1.10 Shamir Threshold Secret Sharing reconstruction algorithm

Input

- List of shares $(j, f(j))$ for j= $0, \dots, t-1$.

Operation

- Compute Lagrange basis polynomials l_j for $j = 0, \dots, t-1$.
- Reconstruct the polynomial $f(x)$ as $\sum_{j=0}^{t-1} f(j) \cdot l_i(x)$.
- Recover the secret as $s = f(0)$.

Output

- Secret *s*.

9.1.11 Random value generation

Input

- The object type (character, digital number, bytes or other)
- The length “n” of the object

Operation

Here we treat each object type in a different way:

- In case the object type is bytes (or bits, if required); the Secure Random objects already provide methods to return random bytes (or bits).
- In case the object type is not bytes, do the following:
    1. Define *m* to be the possible values that the object can take (for example, in case the object type is a numeric digit, *m=10*).
    2. Map every possible value that the object can take to a number between 0 and m-1.
    3. Compute a random number r between 0 and mn-1. To do that:
        - Find the integer *i* such that $2^{i-1} - 1 < m^n - 1 \leq 2^i - 1$, generate *i* random bits and convert those bits to an **unsigned** integer.
        - If the resulting unsigned integer is strictly greater than m^n-1, go to the previous step.
        - Otherwise, output such integer.
    4. Let $r_0, r_1, \dots, r_{n-1}$ be the base m representation of r.

5. Undo the map for each r_i.
6. Output the resulting symbols.

Output

- An object of the specified type and length.

9.1.12 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 $q - 1$ using the Random value generation primitive.
- 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
    1. $pk_n = pk_n + \ pk_{n+1} + pk_{n+2} + \cdots + \ pk_k$ *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 $(C_0, C_1, \dots, C_n)$

9.1.13 ElGamal decryption

Input

- The mathematical group defined by (p, q, g)
- Private key $(sk_1, sk_2, \dots, sk_k)$ (array of any length $k > 0$).
- Ciphertext $(C_0, C_1, C_2, \dots, C_n)$ of at most k+1 elements and at least 2 elements.

Operation

- If the size of the ciphertext is smaller than the size of the public key+1, compute

1. $sk_n = sk_n + sk_{n+1} + sk_{n+2} + \cdots + sk_k$ *where the value of* sk_n *on the right is the old one and the value on the left is the one used from this point onwards.*

- For each *i* from 1 to n compute $M_i = (C_0)^{-sk_i} \cdot C_i$ where the operations are done in the mathematical group

Output

- The message $(M_1, \ldots, M_n)$.

9.1.14 Maurer's Unified Proofs Prover

Based on Maurer's framework for unifying zero-knowledge proofs of knowledge [9]

Input

- Mathematical Group
- Function PHI
    1. Number of inputs *r* (number of exponents)
    2. Number of outputs m
    3. Array of base elements $[h_1, h_2, \ldots, h_n]$.
    4. Computation rules, which are specified by:
        - for each *i* from 1 to *m* (i.e., each output), a list of pairs of indexes will be given, where for each pair the first index will be in the range $[1, \ldots, n]$ and the second index will be in the range $[1, \ldots, r]$.
- Input data
    1. An array of m group elements $(C_1, \ldots, C_m)$.
    2. An array of r exponents $(s_1, \ldots, s_r)$.
    3. Auxiliary string data "$Data$"
- Hash function algorithm: SHA2-256/224 [10].

Operation

The operation is divided into three steps:

1. Commit step:
    - pick as many random exponents $a_1, \ldots, a_r$ as the number of secrets received as input.
    - Compute $(B_1, \ldots, B_m) = PHI(a_1, \ldots, a_r)$. The PHI function computation is described at the end of this section.
2. Challenge step: compute $c = Hash(C_1 \| C_2 \| \ldots \| C_m \| B_1 \| \ldots \| B_m \| "Data")$.

3. Answer step: for each exponent picked in the commit step, compute $z_i = a_i + c \cdot s_i$.

The proof which is sent is $(c; z_1, \dots, z_r)$.

Phi function computation

On inputs $(s_1, \dots, s_r)$ the function is computed as follows:

- The computation of the *i-th* output will be computed as:
    - Given the list of pairs $(i_1, j_1), (i_2, j_2), (i_3, j_3), \dots$ (the computation rules)
    - Take the base element with index i_1 and exponentiate it to the input with index j_1. This gives a partial result p_1 (a group element).
    - Similarly, take base element with index i_2 and exponentiate it to the input with index j_2. This gives a partial result p_2 (a group element).
    - Perform this operation with all pairs of indexes. Then, multiply all the partial results. The result of the multiplication is the i-th output.

Output

- $(c; z_1, \dots, z_r)$.

9.1.15 ElGamal Re-encryption

Input

- Encryption Parameters (which contains a mathematical group defined by p, q, g)
- Public Key $= (pk_1, \dots, pk_k)$ (array of of any length k>0).
- Ciphertext $(C_0, C_1, C_2, \dots, C_n)$

Operation

- Generate a random exponent s between 1 and q-1 using the Random value generation primitive.
- Generate the first element of the re-encrypted ciphertext by computing $C'_0 = C_0 \cdot g^s \ mod \ p$.
- If $n - 1 < k$, compute
    - $pk_n = pk_n \cdot pk_{n+1} \cdot pk_{n+2} \cdot \dots \cdot pk_k \ mod \ p$ 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 the elements $(C_1, C_2, \dots, C_n)$ in the ciphertext to be re-encrypted, compute: $C'_i = C_i \cdot {pk_i}^s \ mod \ p$. The new value for pk_n should be used if this applies.

Output

- The generated random exponent s

- The re-encrypted ciphertext $(C'_0, C'_1, C'_2, \dots, C'_n)$

9.1.16 ElGamal ciphertexts permutation

Input

- List of elements $\{C_i\}_{i=1}^{N}$

Operation

- Given the number of elements in the input list, call the Permutation generation**.** The output is the permutation: $\vec{a} = \{\pi(1), \dots, \pi(N)\}$
- Construct the output list in the following way: for each element in the permutation array $\pi(i)$:
  - Take the element of the input list $\{C_i\}_{i=1}^{N}$ that is in the position indicated by $\pi(i)$.
  - Set the element in the output list.

Output

- List of permuted elements $\{C_{\pi(i)}\}_{i=1}^{N}$
- Permutation $\vec{a} = \{\pi(1), \dots, \pi(N)\}$

9.1.17 Permutation generation

Input

- Number of elements to be permuted N

Operation

- Generate an array with as many elements as the number of elements received as input. The value of each position is the position itself: $array = [0,1,2 \dots, N-1]$
- Permute the values of the array computed in the previous step:
  - From $i = N-1$ to $i = 0$
    - Select a random integer from 0 (inclusive) to i (exclusive): $randomIndex$
    - Swap the values in positions $randomIndex$ and i

Output

- Permutation $\vec{a} = \{\pi(1), \dots, \pi(N)\} = \{\pi(0), \dots, \pi(N-1)\}$ (notice that we use the same notation that is used in the generation of the proof of s shuffle)

9.1.18 Symmetric key generation

Input

- Key length: 128 bits.
- Provider SunJCE / Forge.

Operation

- Call the Random value generation primitive with input object type = “bytes” and length = key length (depends on the symmetric encryption algorithm used)

Output

- Symmetric Key

9.1.19 Message Authentication Code generation

Input

- bytearray representing the message to be authenticated
- MAC symmetric key
- Algorithm parameters: HMAC SHA2-256 [10] [11]
- Provider SunJCE/Forge

Operation

- Generate the authentication data of the message using the MAC algorithm and provider specified and the key provided.

Output

- Message MAC

9.1.20 Key Derivation Function: KDF1 specification

Defined in ISO-18033-2 and in PKCS#1v2.2 [5] with the name MGF1.

Input

- Bytearray to be derived
- Algorithm parameters: SHA2-256 [10]
- Provider BouncyCastle/Forge
- Output length b

Operation

- Given as input a bytearray *x*, a desired output length *b* in bits and a hash algorithm with output length *hLen*, the primitive works as follows:
    - Let *T* be the empty array

- Define $k = ceil\left(b/_{hLen}\right)$
- For *i* from *0* to *k-1*
  - Convert *i* to a byte array of 4 bytes *I*
  - Hash the concatenation of *x* and *I*
  - Concatenate the hash to *T*

- Output the first *b* bits of *T*

Output

- The derived value

9.1.21 Password-based key derivation function

Input

- Password
- Salt
- Iterations: 32000
- Key length: 128 bits
- PBKDF2 with HMAC-SHA256

Operation

- Derive the symmetric key from the input password, using the input salt, PBKDF algorithm, number of iterations and provider and PBKDF output length (given by the “key length”).

Output

- Symmetric Key

9.1.22 Hash generation

Input

- Data
- Algorithm parameters: SHA2-256 [10]
- Provider SUN/Forge

Operation

- Perform a hash of the data provided using the hash algorithm and provider specified in the internal parameters.

Output

- Hash on input data

9.1.23 Digital signature generation

Input

- Bytearray representing the message to sign
- Signing private key
- Algorithm parameters: signature algorithm RSA with PSS padding and SHA2-256 [10].
- Provider BouncyCastle / Forge / JJWT and Nimbus JOSE (for signatures in JSON).

Operation

- Sign the message using the signing private keys, and the signing algorithm and provider specified.

Output

- Message signature

9.1.24 Symmetric encryption

Input

- Receives a byte array representing the message to encrypt
- Encryption Symmetric key
- Algorithm parameters: AES GCM [12] [13] with 128 bits' key.
- Provider BouncyCastle / Forge.

Operation

- Generates a random IV using the Random value generation primitive with the following input: BYTE type, IV length (given by the encryption algorithm).
- Encrypts the message using the symmetric key, the IV and the encryption algorithm and provider specified.

Output

- Encrypted message concatenated with the IV

9.1.25 Symmetric decryption

Input

- Byte array representing the message to decrypt concatenated with the IV
- Encryption symmetric private key
- Algorithm parameters: AES GCM [12] [13] with 128 bits' key.

- Provider BouncyCastle/Forge

Operation

- Decrypt the message using the symmetric key, *IV* and the encryption algorithm and provider specified.

Output

- Decrypted message /nOK

9.1.26 Group element generation

Input

- Mathematical group (p, q, g)

Operation

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

Output

- The group element H

9.1.27 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: $com_{ck}(\vec{a}; r)$

9.1.28 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 (generate it using the Random value generation primitive) and construct the vector $\vec{a}_0$.
2. Generate the following random elements between 1 and q-1 (generate it using the Random value generation primitive): $r_0 \leftarrow \mathbb{Z}_q$ 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 $\vec{a}, r, b, s, \tau$

Verification

Check that:

- $c_{A_0}, c_{B_0}, \dots, c_{B_{2m-1}} \in \mathbb{G}$
- $E_0, \dots, E_{2m-1} \in \mathbb{H}$
- $\vec{a} \in \mathbb{Z}_q^n$ and $r, b, s, \tau \in \mathbb{Z}_q$

Accept if:

$$c_{B_m} = com_{ck}(0; 0) \text{ and } E_m = C$$

$$c_{A_0}\vec{c}_A^{\vec{x}} = com_{ck}(\vec{a}; r)$$

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

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

9.1.29 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:

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

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

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 (generate it using the Random value generation primitive)
   - List of elements to be committed: $\vec{b}$
   - Commitment key $ck = (G_1, \dots, G_n, H)$

   Obtain the commitment $c_b = com_{ck}(\prod_{j=1}^{m} a_{1j}, \dots, \prod_{j=1}^{m} a_{nj}; s)$

3. Engage in a Hadamard product argument given as input $\vec{c}_A, c_b, \vec{b}, \vec{a}_1, \dots, \vec{a}_m, \vec{r}, s$ (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:
    - $b^{SVPArg} = b$
    - $\vec{a}^{SVPArg} = \vec{b}$
    - $c_a^{SVPArg} = c_b$

Verification

Accept if $c_b \in \mathbb{G}$ and both the Hadamard product argument and the Single value argument are convincing.

9.1.30 Hadamard product argument

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.
- $c_b = com_{ck}(\vec{b}; s)$
- $\vec{b}$
- $\vec{a}_1, \dots, \vec{a}_m$ (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)
- 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 (generate it using the Random value generation primitive)
  - 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:

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

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:

    - $\vec{c}_A^{0Arg} = \left(c_{A_1}^{0Arg}, c_{A_2}^{0Arg}, \ldots, c_{A_m}^{0Arg}\right) = (c_{-1}, c_{A_2}, \ldots, c_{A_m})$ (if $m = 1$ this vector has only two elements (c_{-1}, c_{A_2})).
    - $\vec{c}_B^{0Arg} = \left(c_{B_0}^{0Arg}, c_{B_1}^{0Arg}, \ldots, c_{B_{m-1}}^{0Arg}\right) = (c_D, c_{D_1} \ldots, c_{D_{m-1}})$ (if $m = 1$ this vector has only two elements (c_D, c_{D_1})).
    - $A^{0Arg} = \left(\vec{a}_1^{0Arg}, \vec{a}_2^{0Arg}, \ldots, \vec{a}_m^{0Arg}\right) = (-\vec{1}, \vec{a}_2, \ldots, \vec{a}_m)$ and

      $\vec{r}^{0Arg} = \left(r_1^{0Arg}, \ldots, r_m^{0Arg}\right) = (0, r_2, \ldots, r_m)$ (if $m = 1$ these vectors have only two elements $(-\vec{1}, \vec{a}_2)$,$(0, r_2)$)
    - $B^{0Arg} = (\vec{b}_0^{0Arg}, \vec{b}_1^{0Arg}, \ldots, \vec{b}_{m-1}^{0Arg}) = (\vec{d}, \vec{d}_1, \ldots, \vec{d}_{m-1})$ and

      $\vec{s}^{0Arg} = \left(s_0^{0Arg}, \ldots, s_{m-1}^{0Arg}\right) = (t, t_1, t_2, \ldots, t_{m-1})$ (if $m = 1$ these vectors have only two elements $(\vec{d}, \vec{d}_1)$,(t, t_1))

Verification

Check that:

- $c_{B_2}, \ldots, c_{B_{m-1}} \in \mathbb{G}$
- $c_{B_1} = c_{A_1}$
- $c_{B_m} = c_b$

and define:

$$c_{D_i} = c_{B_i}^{x^i} \quad c_D = \prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i} \quad c_{-1} = com_{ck}(-\vec{1}; 0)$$

Accept if the zero argument is valid.

9.1.31 Zero argument

Input

- $\vec{c}_A = com_{ck}(A; \vec{r})$
- $\vec{c}_B = com_{ck}(B; \vec{s})$
- $(\vec{a}_1, \ldots, \vec{a}_m)$ (the rows of matrix A. Notice that in case $m = 1$ this vector contains 2 elements according to that explained in step 19).
- $\vec{r} = (r_1, \ldots, r_m)$ (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 (generate it using the Random value generation primitive) 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 (generate it using the Random value generation primitive)
   - 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 (generate it using the Random value generation primitive) 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 (generate it using the Random value generation primitive)
   - 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 (generate it using the Random value generation primitive) 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 $\vec{a}, r, \vec{b}, s, t$

Verification

Accept if:

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

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

9.1.32 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 using the Random value generation primitive 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 (generate it using the Random value generation primitive)
   - 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 using the Random value generation primitive.

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

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

7. Commit to the elements generated in the previous steps using the Commitment generation primitive with the following inputs:

- A random exponent $s_1 \in \mathbb{Z}_q$ between 1 and q-1 (generate it using the Random value generation primitive)
- List of elements to be committed $(-\delta_2 - a_2\delta_1 - b_1d_2, \dots, -\delta_n - a_n\delta_{n-1} - b_{n-1}d_n)$
- Commitment key $ck = (G_1, \dots, G_{n-1}, H)$

The result is the commitment $c_\delta = com_{ck}(-\delta_1 d_2, \dots, -\delta_{n-1}d_n; s_1)$

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 $i = 1, \dots, n-1$:

$$\begin{gathered} -\delta_2 - a_2\delta_1 - b_1d_2 \\ -\delta_3 - a_3\delta_2 - b_2d_3 \\ \vdots \\ -\delta_{i+1} - a_{i+1}\delta_i - b_id_{i+1} \\ \vdots \\ -\delta_n - a_n\delta_{n-1} - b_{n-1}d_n \end{gathered}$$

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 (generate it using the Random value generation primitive)
   - List of elements to be committed:$(-\delta_2 - a_2\delta_1 - b_1d_2, \dots, -\delta_n - a_n\delta_{n-1} - b_{n-1}d_n)$
   - Commitment key $ck = (G_1, \dots, G_{n-1}, H)$

   The result is the commitment $c_\Delta = com_{ck}(\delta_2 - a_2\delta_1 - b_1d_2, \dots, \delta_n - a_n\delta_{n-1} - b_{n-1}d_n; s_x)$
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:

$$\begin{gathered} \tilde{a}_1 = xa_1 + d_1 \\ \vdots \\ \tilde{a}_n = xa_n + d_n \\ \tilde{r} = xr + r_d \end{gathered}$$

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

$$\begin{gathered} \tilde{b}_1 = xb_1 + \delta_1 \\ \vdots \\ \tilde{b}_n = xb_n + \delta_n \\ \tilde{s} = xs_x + s_1 \end{gathered}$$

Output

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

Verification

Accept if:

- $c_d, c_\delta, c_\Delta \in \mathbb{G}$
- $\tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_n, \tilde{b}_n, \tilde{r}, \tilde{s} \in \mathbb{Z}_q$
- and the following equations hold:

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

$$\tilde{b}_1 = \tilde{a}_1 \quad \tilde{b}_n = xb$$

9.2 Optimizations at the voting client context in the voting phase

Several optimizations can be done to reduce the time needed to cast a vote. They are focused on reducing the number of modular exponentiations to be computed once the voter is ready to send his/her vote. The modular exponentiations take much more time than any other operation executed by the voting client, and therefore is where the optimization efforts are focused.

Three strategies can be followed:

- **Pre-computation at the voting client**: Some modular exponentiations can be computed while the voter is navigating through the application, and before the voter clicks on the “send” button.
- **Pre-computation at configuration**: Some modular exponentiations are already being computed during the configuration. Their results can be stored in the password-sealed KeyStore sent to the voter at the voting phase, so that they can be recovered, and used to construct the vote to be sent.
- **Use of short exponents**: For some operations, exponents may be shorter than what is required by the mathematical group where the operations are done, without posing a security risk. In [14] it is demonstrated that full exponents and short exponents are indistinguishable under the Discrete Logarithm with Short Exponent (DLSE) assumption, and that there is no efficient algorithm which solves the DLSE problem with non-negligible probability. Currently this optimization is not used.

9.2.1 Pre-computation at the Voting Client

Pre-computation operations may be done at the voting client by using “multithreading” features given by web workers. Another alternative is that they are computed between page changes, etc.

There are two kinds of values that can be pre-computed at the voting client: ones which do not depend on the voter opening her KeyStore (that is, entering the Start Voting Key (SVK_{id})), and ones which do depend on that. Here we provide a specification of how to do pre-computations of both values:

Preconditions for pre-computations:

- Execute the entropy collector when the first page loads, set to an entropy value of 256 bits. When it stops, the application is ready to start generating random values. Precomputations must

not start before that. If the voter clicks on “send” and the entropy collector has not already stopped (there is no initialized “prng”), stop it and proceed to do the computations.

- Retrieve the encryption parameters, the Election public key (EL_{pk}) and the Choice Return Codes encryption public key after the voter has been authenticated (after the voter receives the authentication token, together with the election and ballot-related information.
- Open the voter *Verification Card KeyStore* and retrieve the *Verification Card Secret Key*.

Pre-computations:

1. Voting options encryption
    a. Use short exponent for encryption.
    b. Precompute encryption values with election public key.
2. Partial Choice Return Codes encryption.
    a. Use short exponent for encryption.
    b. Precompute encryption values with partial Choice Return Codes encryption public key.
3. Cryptographic proofs generation.
    a. Schnorr proof: can be fully computed after step 1 is ready (only C0, the first part of the precomputed values for the vote encryption, is needed). Short exponents can be used.
    b. Exponentiation proof generator: use the pre-compute method to pre-compute part of the exponentiations.
    c. Plaintext equality proof generator: use the pre-compute method to pre-compute part of the exponentiations.
    d. Exponentiated ciphertext pre-computation: raise the pre-computed encryption values from step 1 to the verification card private key (k_{id}).

Operations that can be done after a voter selection:

4. Partial Choice Return Codes computation: raise the selected voting option to the verification card private key (k_{id}).

Using the pre-computed values after voter finishes selections:

5. Voting options encryption: multiply together all the selected voting options and multiply the result by the second part (the phi) of the pre-computed values in step 1.
6. Partial Choice Return Codes encryption.
    a. Multiply each partial Choice Return Code computed in step 4 to one of the phi components of the pre-computed encryption values in step 2.
7. Cryptographic proofs generation.

a. Exponentiation proof generator: use the generate method from the proof, using the values pre-computed in step 3.

b. Plaintext equality proof generator: use the generate method from the proof, using the values pre-computed in step 3.

c. Exponentiated ciphertext pre-computation: Multiply together all the partial Choice Return Codes computed in step 4 (a multiplication / compression in the mathematical group). Multiply the result by the second part (the phi) of the values pre-computed in step 3.c.

9.3 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.

9.3.1 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 Voter Portal Frontend: Frontend application that guides the voters throughout all the voting steps enabling them to successfully cast a vote for a particular election.

9.3.2 Copyright notice

9.3.2.1 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.

9.3.2.2 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).

9.3.2.3 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.