



Swiss Post Voting System

System Specification

Swiss Post*

Version 1.5.0

Abstract

The Swiss Post Voting System is a return code-based remote online voting system that provides individual verifiability, universal verifiability, and vote secrecy. This document provides a detailed specification of the cryptographic protocol—from the configuration phase to the voting phase to the counting phase—and includes pseudocode representations of most algorithms. The document specifies the protocol’s higher-level algorithms as well as some of the underlying building blocks.

* Copyright 2025 Swiss Post Ltd. - Based on original work by Scytel Secure Electronic Voting S.A. (succeeded by Scytel Election Technologies S.L.U.), modified by Swiss Post. Scytel Election Technologies S.L.U. is not responsible for the information contained in this document.

Disclaimer

E-Voting Community Program Material - please follow our [Code of Conduct](#) describing what you can expect from us, the Coordinated Vulnerability Disclosure Policy, and the contributing guidelines.

Revision chart

Version	Description	Author	Reviewer	Date
0.9.5	Initial published version	OE	XM, JS, TH	2021-05-04
0.9.6	See change log for version 0.9.6	OE	XM, JS, TH	2021-06-25
0.9.7	See change log for version 0.9.7	OE	XM, JS, TH	2021-10-15
0.9.8	See change log for version 0.9.8	OE	XM, JS, TH	2022-02-17
0.9.9	See change log for version 0.9.9	HR, OE	XM, JS, TH	2022-04-18
1.0.0	See change log for version 1.0.0	HR, OE, TH	XM, JS	2022-06-24
1.1.0	See change log for version 1.1.0	HR, OE, TH	XM, JS	2022-10-03
1.1.1	See change log for version 1.1.1	HR, OE, TH	XM, JS	2022-10-31
1.2.0	See change log for version 1.2.0	AH, OE	XM, JS, TH	2022-12-09
1.3.0	See change log for version 1.3.0	AH, OE	XM, JS, TH	2023-04-19
1.3.1	See change log for version 1.3.1	AH, OE	XM, JS, TH	2023-06-15
1.3.2	See change log for version 1.3.2	AH, OE	XM, JS, CK	2023-10-23
1.4.0	See change log for version 1.4.0	AH, CC, CK, OE	XM, JS	2024-02-14
1.4.1	See change log for version 1.4.1	AH, CC, CK, OE	XM, JS	2024-06-17
1.4.1.1	See change log for version 1.4.1.1	CC, OE	CK, PO	2024-07-30
1.4.2	See change log for version 1.4.2	AH, CC, CK, OE	XM, JS	2025-05-16
<u>1.5.0</u>	<u>See change log for version 1.5.0</u>	<u>AH, CC, CK, OE</u>	<u>XM, JS</u>	<u>2025-06-20</u>

Contents

Symbols	6
1 Introduction	9
1.1 Two-Round Return Code Scheme	10
1.2 Security Objectives	14
1.3 Limitations	15
1.4 Conventions	16
1.5 Context, State, and Input Variables	18
1.5.1 Context Variables	18
1.5.2 Input Variables	19
1.5.3 Stateful Lists and Maps	19
2 System Overview	20
2.1 Voters	20
2.2 Voting Client	20
2.3 Voting Server	20
2.4 Control Components	21
2.5 Electoral Board	21
2.6 Setup Component	21
2.7 Printing Component	22
2.8 Tally Control Component	22
2.9 Auditors	22
2.10 Dispute Resolver	22
3 Preliminaries	23
3.1 Basic Data Types	23
3.2 Cryptographic Parameters and System Security Level	23
3.3 Keys and Codes of the Voting Card	24
3.4 Election Event Context	25
3.4.1 Data Model	25
3.4.2 Hashing the Election Event Context	26
3.4.2 Execution Context of Algorithms	26
3.5 Electoral Model	28
3.5.1 Electoral Model and Parameters	28
3.5.2 Encoding of Voting Options	30
3.5.3 Primes Mapping Table	31
3.5.4 Valid Combinations of Voting Options	38
3.5.5 Hashing the Voting Options' Context	40
3.5.5 Multiplying and Factorizing Voting Options	40
3.6 Agreement Algorithms	42
3.6.1 Agreement on the Global Election Event Context	42
3.6.2 Agreement on the Verification Card Set-Specific Data	44
3.6.3 Agreement on the Entire Election Event-Specific Data	45
3.6.4 Agreement on the Sent Votes from the Voting Phase	48
3.6.5 Proof of Correct Key Generation	49
3.7 Voter Authentication	52
3.8 Write-Ins	54
3.8.1 Alphabet used for Write-Ins	55

3.8.2	Encoding Write-Ins	55
3.8.3	Decoding Write-Ins	57
3.8.4	Creating a Vote with Write-Ins	58
3.8.5	Processing Write-Ins in the Tally Phase	59
3.9	Proof of Correct Key Generation	61
4	Configuration Phase	61
4.1	SetupVoting	63
4.1.1	GenSetupData	64
4.1.2	GenVerDat	66
4.1.3	GetVoterAuthenticationData	67
4.1.4	GenKeysCCR	68
4.1.5	GenEncLongCodeShares	69
4.1.6	CombineEncLongCodeShares	71
4.1.7	GenCMTable	75
4.1.8	GenVerCardSetKeys	76
4.1.9	GenCredDat	77
4.2	SetupTally	78
4.2.1	SetupTallyCCM	79
4.2.2	SetupTallyEB	80
4.3	<u>Finalize Configuration Phase</u>	82
5	Voting Phase	84
5.1	AuthenticateVoter	85
5.1.1	GetAuthenticationChallenge	87
5.1.2	VerifyAuthenticationChallenge	88
5.1.3	GetKey	90
5.2	SendVote	92
5.2.1	CreateVote	94
5.2.2	VerifyBallotCCR	96
5.2.3	PartialDecryptPCC	97
5.2.4	DecryptPCC	99
5.2.5	CreateLCCShare	100
5.2.6	ExtractCRC	102
5.3	Confirm Vote <u>ConfirmVote</u>	103
5.3.1	CreateConfirmMessage	104
5.3.2	CreateLVCCShare	105
5.3.3	VerifyLVCCHash	106
5.3.4	ExtractVCC	107
6	Tally Phase	109
6.1	MixOnline	111
6.1.1	GetMixnetInitialCiphertexts	113
6.1.2	VerifyMixDecOnline	114
6.1.3	MixDecOnline	115
6.1.4	Handling Inconsistent Views of Confirmed Votes	117
6.2	<u>Handling Inconsistent Views of Confirmed Votes</u>	117
6.2.1	<u>Overview and Trigger of the Dispute Resolution Process</u>	117
6.2.2	<u>Extraction Phase</u>	119
6.2.3	<u>Dispute Resolver's Consistency Checks</u>	119

6.2.4	<u>Control Component's Update</u>	124
6.3	MixOffline	125
6.3.1	VerifyVotingClientProofs	125
6.3.2	VerifyMixDecOffline	126
6.3.3	MixDecOffline	127
6.3.4	ProcessPlaintexts	128
6.3.5	<u>Creating the Tally FilesFile</u>	130
6.3.6	Requesting a Proof of Non-Participation	131
7	Channel Security	133
7.1	<u>Digital Signatures for Protocol Messages</u>	133
7.2	<u>Digital Signatures for File Interfaces (XML)</u>	139
7.3	<u>Streamable Symmetric Encryption and Decryption for Dataset Security</u>	143
	References	147
	List of Algorithms	149
	List of Figures	151
	List of Tables	151

Symbols

\mathbb{A}_{10}	Decimal numbers
\mathbb{A}_{Base16}	Base16 (Hex) alphabet [15]
\mathbb{A}_{Base32}	Base32 alphabet, including the padding character = [15]
\mathbb{A}_{Base64}	Base64 alphabet, including the padding character = [15]
\mathbb{A}_{UCS}	Alphabet of the Universal Coded Character Set (UCS) according to ISO/IEC10646
\mathbb{A}_{latin}	Extended Latin alphabet, as described in the crypto primitives specification
\mathbb{A}_{u32}	User-friendly alphabet for codes, as described in the crypto primitives specification
\mathcal{T}_1^{50}	Alphabet used for voting option identifiers (word characters and minus sign: $[\backslash \mathfrak{w} \backslash -] \{1, 50\}$)
aux	Auxiliary string
\mathcal{B}	Set of possible values for a byte
\mathcal{B}^*	Set of byte arrays of arbitrary length
\mathbb{B}^n	Set of bit arrays of length n
bb	Ballot box ID
BCK	Ballot Casting Key
CC	Short Choice Return Codes
\mathbf{c}_{ck}	Encrypted confirmation keys
\mathbf{c}_{Dec}	List of partially decrypted ciphertexts
\mathbf{c}_{expCK}	Exponentiated, encrypted, hashed Confirmation Keys
\mathbf{c}_{expPCC}	Exponentiated, encrypted, hashed partial Choice Return Codes
\mathbf{c}_{mix}	List of shuffled ciphertexts
\mathbf{c}_{pCC}	Encrypted, hashed partial Choice Return Codes
\mathbf{c}_{pC}	Encrypted pre-Choice Return Codes
CK	Confirmation Key
CCM	Mixing control components
CCR	Return Codes control components
\mathbf{d}	Exponentiated γ elements of the encrypted Partial Choice Return Codes: $(d_0, \dots, d_{\psi-1})$
δ	Number of write-in options + 1 for a specific verification card set
δ_{max}	Maximum number of write-in options + 1 across all verification card sets
δ_{sup}	Maximum supported number of write-in options + 1
EA	Extended authentication factor
E1	Encrypted vote
$\widetilde{E1}$	Exponentiated encrypted vote
E2	Encrypted partial Choice Return Codes
$\widetilde{E2}$	Multiplied, encrypted partial Choice Return Codes
ee	Election event id
g	Generator of the encryption group

γ	Gamma - the ElGamal ciphertext's first element
\mathbb{G}_q	Group of quadratic residues modulo p of size q
hAuth	Base authentication challenge
hCK	Hashed, squared Confirmation Key
hpCC	Hashed, squared partial Choice Return Codes
id	Voter index $\{0, \dots, N_E - 1\}$
j	Index over control components $\{1, 2, 3, 4\}$
lCC	Long Choice Return Codes
lVCC	Long Vote Cast Return Code
l_{BCK}	Character length of ballot casting keys
l_{CC}	Character length of Choice Return Codes
l_{ID}	Character length of unique identifiers: 32
l_{HB64}	Character length of the Base64 encoded hash function output
l_{VCKS}	Character length of the Base64 encoded verification card keystore: 572
l_{KD}	Output byte length of the key derivation function: 32
l_{EA}	Character length of the extended authentication factor
l_{SVK}	Character length of Start Voting Keys
l_{VCC}	Character length of Vote Cast Return Codes
l_w	Maximum number of characters in a write-in field: 400
L_{lVCC}	Long Vote Cast Return Codes allow list
L_{pCC}	Partial Choice Return Codes allow list
L_{votes}	List of decrypted votes
L_{decodedVotes}	List of decoded votes
L_{writelns}	List of decoded write-ins
m	Message representing the vote
m	List of plaintext votes
N	Set of positive integer numbers including 0
N⁺	Set of strictly positive integer numbers
N_{bb}	Number of ballot boxes of an election event
N_S	Number of sent votes in a specific ballot box
N_C	Number of confirmed votes in a specific ballot box
N̂_C	Number of mixed votes including trivial encryptions
N_E	Number of eligible voters of a specific verification card set
n	Number of voting options for a given verification card set
n_{max}	Maximum number of voting options across all verification card sets
n_{sup}	Maximum supported number of voting options
ϕ	Phi - the ElGamal ciphertext's second element
ψ	Allowed number of selections for a given verification card set
ψ_{max}	Maximum number of selections across all verification card sets
ψ_{sup}	Maximum supported number of selections

$\pi_{\text{decPCC},j}$	Proof of correct decryption of the partial Choice Return Codes
π_{Exp}	Proof of correct exponentiation
π_{EqEnc}	Proof of equality of encryptions (plaintext equality)
pCC	Partial Choice Return Codes
pC	pre-Choice Return Codes
pVCC	pre-Vote Cast Return Codes
$\tilde{\mathbf{p}}$	Encoded voting options $(\tilde{p}_0, \dots, \tilde{p}_{n-1})$, $\tilde{p}_k \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$
$\tilde{\mathbf{p}}_{\text{w}}$	Encoded voting options corresponding to the choice of a write-in $(\tilde{p}_{\text{w},0}, \dots, \tilde{p}_{\text{w},\delta-2})$, $\tilde{p}_{\text{w},k} \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$
$\hat{\mathbf{p}}$	Selected encoded voting options $(\hat{p}_0, \dots, \hat{p}_{\psi-1})$, $\hat{p}_k \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$
\mathbb{P}	Set of prime numbers
\mathbf{p}	Vector of small prime group members
p	Encryption group modulus
q	Encryption group cardinality s.t. $p = 2q + 1$
$\hat{\mathbf{s}}$	Selected write-in candidates
SVK	Start Voting Key
σ	Semantic information $(\sigma_0, \dots, \sigma_{n-1})$, $\sigma_k \in \mathbb{A}_{UCS}^*$
τ	Correctness information $(\tau_0, \dots, \tau_{n-1})$, $\tau_k \in \mathcal{T}_1^{50}$
$\hat{\tau}$	Blank correctness information $(\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1})$, $\hat{\tau}_k \in \mathcal{T}_1^{50}$
$\tilde{\mathbf{v}}$	Actual voting options (v_0, \dots, v_{n-1}) , $v_k \in \mathcal{T}_1^{50}$
$\hat{\mathbf{v}}_{\text{id}}$	Selected actual voting options $(\hat{v}_0, \dots, \hat{v}_{\psi-1})$, $\hat{v}_k \in \mathcal{T}_1^{50}$
vc	Verification card
\mathbf{vc}	Vector of verification card IDs of an election event
VCks	Verification card keystore
vcs	Verification card set ID
\mathbf{vcs}	Vector of verification card set IDs of an election event ordered lexicographically
VCC	Short Vote Cast Return Code
vcd	Voting card
$ x $	Bit length of the number x
\mathbf{w}_{id}	Voter's encoded write-ins $(w_{\text{id},0}, \dots, w_{\text{id},\delta-2})$
\mathbb{Z}_q	Group of integers modulo q
\top	Truth value true or successful termination
\perp	Truth value false or unsuccessful termination

1 Introduction

Switzerland has a longstanding tradition of direct democracy, allowing Swiss citizens to vote approximately four times a year on elections and referendums. In recent years, voter turnout hovered below 40 percent [11].

The vast majority of voters in Switzerland fill out their paper ballots at home and send them back to the municipality by postal mail, usually days or weeks ahead of the actual election date. Remote online voting (referred to as e-voting in this document) would provide voters with some advantages. First, it would guarantee the timely arrival of return envelopes at the municipality (especially for Swiss citizens living abroad). Second, it would improve accessibility for people with disabilities. Third, it would eliminate the possibility of an invalid ballot when inadvertently filling out the ballot incorrectly.

In the past, multiple cantons offered e-voting to a part of their electorate. Many voters would welcome the option to vote online - provided the e-voting system protects the integrity and privacy of their vote [12].

State-of-the-art e-voting systems alleviate the practical concerns of mail-in voting and, at the same time, provide a high level of security. Above all, they must display three properties [18]:

- Individual verifiability: allow a voter to convince herself that the system correctly registered her vote
- Universal verifiability: allow an auditor to check that the election outcome corresponds to the registered votes
- Vote secrecy: do not reveal a voter's vote to anyone

Following these principles, the Federal Chancellery defined stringent requirements for e-voting systems. The Ordinance on Electronic Voting (VEleS - Verordnung über die elektronische Stimmabgabe) and its technical annex (VEleS annex) [7] describes these requirements.

Swiss democracy deserves an e-voting system with excellent security properties. Swiss Post is thankful to all security researchers for their contributions and the opportunity to improve the system's security guarantees. We look forward to actively engaging with academic experts and the hacker community to maximize public scrutiny of the Swiss Post Voting System.

1.1 Two-Round Return Code Scheme





The Swiss Post Voting System is a return code scheme: every voter receives a printed code sheet prior to the election. Figure 1 shows an example code sheet, which contains the following types of codes (the codes are unique for every voter and every election event):


- A Start Voting Key to start the voting process
- A Choice Return Code for each voting option
- A Ballot Casting Key to confirm the vote
- A Vote Cast Return Code

Voting Card - E-Voting

To submit your vote electronically, please follow the steps in the enclosed instructions.

E-Voting Portal: `xy.evoting.ch`

	Start Voting Key <code>bmsr 6abc 8enw js3 5uc7 87ex</code>
	Choice Return Codes See further below
	Ballot Casting Key <code>6423 6545 8</code>
	Vote Cast Return Code <code>5847 5857</code>

 **Choice Return Codes**

Popular Vote


Question 1		
Yes: 6731	No: 1186	Blank: 3816
Question 2		
Yes: 4832	No: 9175	Blank: 2406

Fig. 1: ~~Printed~~ Example of printed code sheet sent to the voters. Codes are unique per voter and election event.

Moreover, the voter receives instructions by mail that help ensure the voting process is followed correctly and allow the detection of any attempts by a malicious voting portal to mislead the voter into deviating from the expected procedure. Figure 2 shows example voting instructions.

Instructions for Electronic Voting


The voting card contains your access data and security elements for electronic voting. Please keep it safe and secure. The information on the voting card and in this guide takes precedence over all other information.

 **Start the E-Voting Portal**

Enter xy.evoting.ch directly into your browser's address bar

1. Review explanations and legal provisions

Read the explanations and legal provisions and confirm that you have reviewed them.

 **2. Start the Voting Process**


Enter your Start Voting Key and date of birth.

3. Enter your vote


Select voting options or leave them blank

4. Verify your vote


Check your vote before encrypting and submitting it.

 **5. Verify Choice Return Codes**


Do the Choice Return Codes shown on the voter portal match those on your voting card? If not, abort the voting process and contact your municipality.

 **6. Enter Ballot Casting Key**

If all Choice Return Codes match, enter the Ballot Casting Key. If you do not enter your Ballot Casting Key, your vote will not be placed in the electronic ballot box.


 **7. Verify Vote Cast Return Code**

Does the Vote Cast Return Code shown on the voter portal match the one on your voting card? If yes, your vote has been placed in the electronic ballot box. If a different code is shown, contact your municipality.

 **Additional Security Notes**

To use the e-voting portal, it is recommended to use a private browser mode (e.g., incognito mode in Google Chrome) without add-ons.

More security notes and instructions can be found at xy.evoting.ch

 **Support**

If you have any questions, errors, or discrepancies related to e-voting, please contact your municipality

Fig. 2: Example of printed instructions sent to the voters.

Figure 3 highlights the two rounds of the voting process from the voter’s perspective.

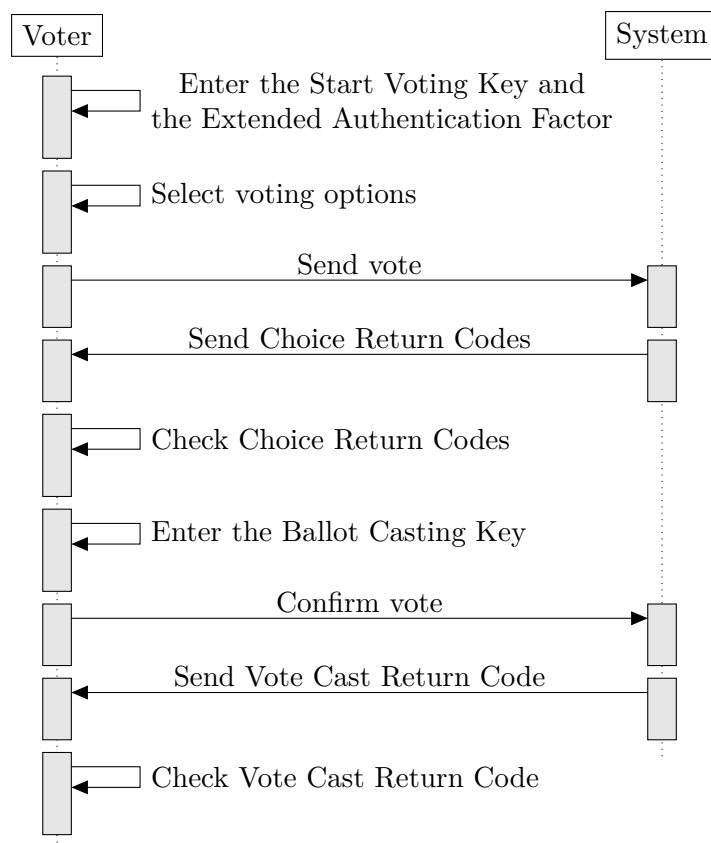


Fig. 3: The voting process in a two-round return code scheme. In the implementation, an authentication step precedes the voter’s voting options selection.

First, the voter enters the Start Voting Key to authenticate herself and selects the candidates she wants to vote for (or answers questions in the case of a referendum). In turn, the system responds with a Choice Return Code for each selected voting option. The voter checks that the Choice Return Codes match the ones printed on the voting card—otherwise, the voter aborts the process and alerts the election authorities. If the Choice Return Codes match, the submitted vote corresponds to the voter’s intention, and the voter enters the Ballot Casting Key. Finally, the system acknowledges a successful confirmation by sending back the Vote Cast Return Code.

If the voter aborts the process after sending or confirming the vote, she can resume the process (potentially on a different voting device) by logging in again.

Figure 4 shows the process after the voter sent the vote in a previous session (but did not confirm it), while figure 5 shows the process after the voter confirmed the vote earlier on.

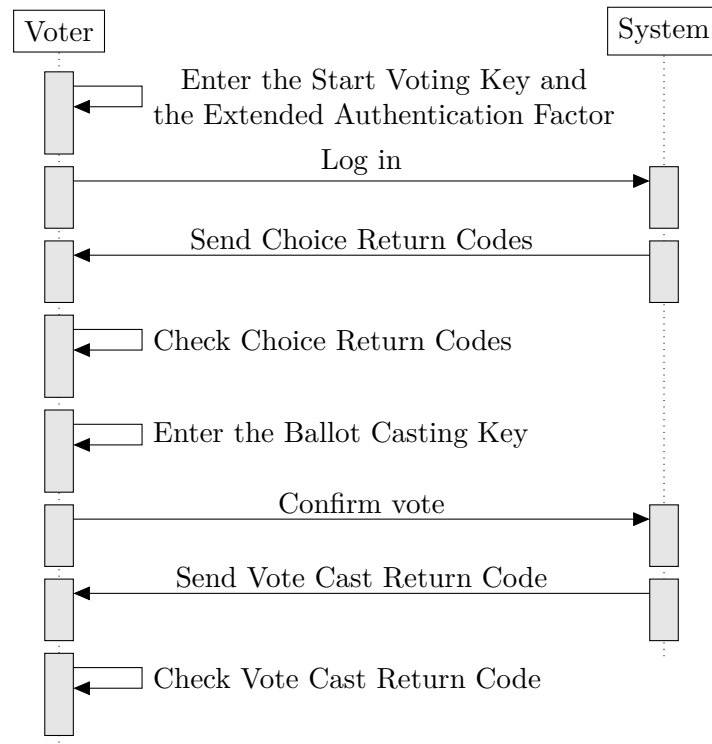


Fig. 4: The voting process when the voter aborted voting in an earlier session after having sent the vote. In the earlier session, the voter did not yet confirm the vote. The Choice Return Codes returned by the system correspond to the selected voting options in the earlier session. Please note that at this stage, the voter can no longer change the selections. However, the vote is not yet considered final, and the voter might still decide to use a different voting channel.

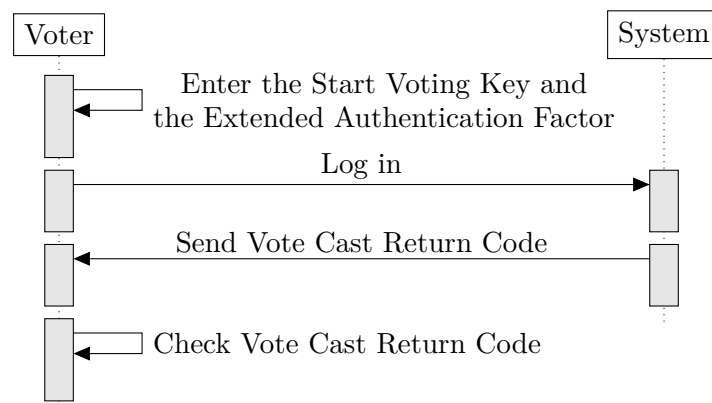


Fig. 5: The voting process when the voter aborted voting in an earlier session after vote confirmation. The voter can no longer change the selections and the vote is considered final. The voter may repeat this process on multiple devices until she convinces herself that the system returns the correct Vote Cast Return Code.

1.2 Security Objectives

The computational proof [20] details the Swiss Post Voting System’s threat model and formally proves the following security properties:

- Individual verifiability
- Universal verifiability
- Vote secrecy

Individual verifiability allows the voter to check that the server correctly registered her vote and contains the voter’s choices. Faithfully executing the two-round return code scheme (see section 1.1) guarantees to the voter that the system’s trustworthy part registered the intended vote. Conversely, this implies that even a powerful adversary, controlling the voting client and most of the server infrastructure, cannot alter or drop a vote without being detected by a diligent voter or auditor.

Universal verifiability allows voters or auditors to check that the system correctly counted all confirmed votes. Independent auditors verify every step in the counting process—from the registration of the encrypted voting options to the decryption and tallying process—without compromising vote secrecy. Advanced cryptographic techniques such as non-interactive zero-knowledge proofs and verifiable mix-nets allow the Swiss Post Voting System to have the best of both worlds: the election result is correct beyond doubt, and every single vote remains secret.

Vote secrecy preserves the privacy of the voter. By extension, vote secrecy ensures that no component learns the election results before the final decryption step. The Swiss Post Voting System protects vote secrecy by encrypting votes end-to-end and splitting the decryption key among multiple entities.

1.3 Limitations

No cryptographic protocol is unconditionally secure. The protocol of the Swiss Post Voting System bases its security objectives on a few assumptions. Even though these assumptions comply with the Federal Chancellery's Ordinance on Electronic Voting [7], we will highlight some of them for the sake of clarity and transparency.

- First, we are defending our security properties against an adversary that is polynomially bounded. An efficient, fault-tolerant quantum computer could break some of the mathematical assumptions underpinning the Swiss Post Voting System. Quantum-resistant e-voting protocols exist, but they are relatively recent, and their security properties are less well understood. We consider quantum-resistance an area for future improvement of the protocol.
- Second, to provide vote secrecy, the Swiss Post Voting System assumes that the attacker is not controlling the voting client. While we guarantee individual verifiability even if the voting client is malicious, we cannot use cryptography to prevent an attacker from spying on the voter's choices on malicious clients. However, the Swiss Post Voting System prevents the server from breaking vote secrecy if at least one control component is honest.
- Third, the Swiss Post Voting System assumes a trustworthy setup and printing component. The setup and printing component generate keys and codes in conjunction with the control components and sends the voters' voting cards. By its nature, the setup and printing component handle sensitive data, and the process for generating and sending voting cards must meet rigorous requirements. In general, e-voting systems aim to distribute trust and detect any attack or malfunction if at least one component works correctly. However, we believe that there are certain limits for distributing the functionality of the setup and printing component: one cannot expect the voter to combine different code sheets by hand, and the costs of printing multiple code sheets in independent printing facilities would currently be prohibitive. For now, we propose to implement a return code scheme with a single setup and printing component, which must put organizational and technical means into place to prevent the leaking of the secret codes.

1.4 Conventions

We use the following conventions throughout the document.

- Display algorithms in font without serif and **vectors** in **boldface**.
- Explicit domains and ranges of context, input, and output variables. We assume that the implementation ensures the correct domain of the input and context elements. E.g. this means that when an input has the expected form $\mathbf{x} = (x_0, \dots, x_{n-1}) \in (\mathbb{G}_q)^n$, the implementation checks that the input elements have the correct form: That \mathbf{x} has exactly n elements and each one is a group member, for instance by calculating that the Jacobi Symbol of each element of \mathbf{x} equals 1.
- Use the terms public key and secret key (instead of private key) and abbreviate them with pk and sk .
- Use 0-based indexing.
- ~~Use nested structures only when explicitly specified.~~ Flatten lists when defined over multiple lines of pseudocode. Therefore, the expression

$$list \leftarrow (a, b)$$

$$list \leftarrow (list, c, d)$$

results in a list without any nested structures:

$$list = (a, b, c, d)$$

- Use ~~the union operator \cup~~ nested structures when defined over one line of pseudocode. Therefore, the expression

$$\underline{list_1 \leftarrow (a, b)}$$

$$\underline{list_2 \leftarrow (c, d)}$$

$$\underline{list_3 \leftarrow (list_1, list_2)}$$

results in a list with nested structures:

$$\underline{list_3 = ((a, b), (c, d))}$$

- Explicitly include the structure of empty lists in the algorithms.

$$\underline{() \neq (())}$$

Hence, we also represent empty lists in nested structures.

- Use the operator \parallel to indicate that we append an element to a list:

$$list \leftarrow (a, b)$$

$$list \leftarrow list \underline{\cup} \parallel c$$

results in

$$list = (a, b, c)$$

- Slightly abuse the \in -notation to check whether an element is part of a list or not:

$$list \leftarrow (a, b)$$

$$a \in list = \top$$

$$c \in list = \perp$$

- Define sets for ranges, e.g. for $i \in [0, n)$. This expression indicates that we include the lower bound but exclude the upper bound, i.e. $0 \leq i < n$.
- For a key-value map L consisting of pairs (**key**, **value**), we define the retrieval of the **value** $\in \mathcal{B}$ corresponding to a **key** $\in \mathcal{A}$ as following:

$$L : \mathcal{A} \rightarrow \mathcal{B}$$

$$L(\text{key}) = \text{value}$$

If the map has more than two entries, we still use the same notation while making it clear which of the entries should be retrieved from the map.

1.5 Context, State, and Input Variables

The specification distinguishes *context* and *input* variables as well as *stateful lists and maps*.

1.5.1 Context Variables

In cryptographic protocols, a context variable acts as a common reference for participants during algorithm execution. The trustworthiness of these variables is derived from their deterministic nature, mutual acceptance among participants, and the capacity for verification against external public data or information from prior protocol stages. As a result, context variables are trusted more than input variables. In algorithms, input variables undergo validation against these context variables. Typically, context variables remain invariant across multiple algorithm invocations. Furthermore, when an algorithm calls another, the callee inherits the context variables from the caller without the need for explicitly passing context variables as parameters. Context variables encompass the following parameters:

- Group parameters
- Election event context identifiers and component indices
- Election event parameters
- Public keys (under certain conditions)
- Static parameters

Group parameters: These parameters can be deterministically derived or validated by protocol participants using a seed (see section 3.2). Consequently, a participant either initially establishes the group parameters independently or obtains them from a trustworthy source.

Election event context identifiers and component indices: Algorithms are executed with reference to a specific entity within the election event context, as indicated in table 7. Protocol participants in the Swiss Post Voting System have a common view of the election event context (see section 3.4). They achieve this common view by receiving the election event context from a trustworthy source, cross-checking it with publicly available information, or integrating it in the challenges of the zero-knowledge proofs (see the algorithms 3.11

[GetHashElectionEventContext](#) and [3.12 GetHashContext](#) and [3.15 GetHashExtractedElectionEvent](#)). This common understanding enables each participant to deduce the corresponding identifiers of other entities within the election event. For example, one might derive the verification card set ID from a verification card ID, or the election event ID from a verification card ID. Therefore, the election event context identifiers typically originate from the component's internal view, thereby constituting a context for the algorithm. If the context of an algorithm requires an entire vector of verification card IDs, the protocol participants must verify that the vector contains the correct verification card IDs in the expected order by matching it with information from earlier protocol steps.

We omit the election event context identifiers in the specification of an algorithm if they are not utilized during the algorithm's operation.

Furthermore, it is imperative for the implementation to validate the election event context. Certain algorithms are restricted to specific protocol phases or conditions. For instance,

re-executing an algorithm from the configuration phase is prohibited once that phase has concluded.

Several algorithms are executed by multiple control components, each possessing a unique index. We assume that each control component knows its respective index, thus we classify the index as a context variable.

Election event parameters: Section 3.5 elaborates on the electoral model, which sets the parameters for various elections and votes. These parameters, such as the maximum number of selections or the total number of candidates, can typically be verified using publicly available information. Moreover, the association of voting options with prime numbers and the proper structure of a submitted vote are also verifiable, as further discussed in Section 3.5.3. Therefore, we consider the parameters of the election event as context variables.

Public keys (under certain conditions): The Swiss Post Voting System distributes the generation of certain public keys and ensures their correct generation with zero-knowledge proofs (see section 3.6.5). When the protocol ensures that a trustworthy component participated in the generation of a public key and the proofs of correct key generation were previously verified, this public key can be treated as a context variable for the execution of a given algorithm. On the other hand, if the public key does not fulfill these conditions, it must be classified as an input variable, requiring extensive verification. Specifically, secret keys and their corresponding key pairs are always classified as input variables, as the private nature of secret keys precludes their inclusion in a possibly shared context.

Static parameters: The section Symbols defines the majority of constants and frequently utilized parameters. Should an algorithm necessitate a particular static parameter, it will be categorized as a context variable, given that this parameter can be extrapolated from established standards or cryptographic best practices.

1.5.2 Input Variables

Any variable not classified as a context variable is deemed an input variable. These input variables, potentially originating from unreliable sources, necessitate comprehensive, multi-layered validation. It is critical, especially when an algorithm employs a variable as both context and input, to prioritize the context variable as the authoritative source. For instance, in scenarios where the context incorporates a variable n and the input includes a list of length n , the implementation must verify that the list's actual size corresponds to the context variable n .

1.5.3 Stateful Lists and Maps

Certain algorithms must be executed only once, or only a specified number of times, or must not be executed before another algorithm. Consequently, maintaining and updating state across multiple algorithm invocations is critical. Broadly, algorithms modifying stateful lists and maps lack idempotency, as repeated executions with identical context and input variables may yield different results.

The implementation must establish a robust synchronization mechanism to reliably manage updates and retrievals involving stateful lists and maps.

2 System Overview

The following sections explain the different actors of the Swiss Post Voting System. The computational proof of the cryptographic protocol elaborates on the threat models for the different security objectives [20].

2.1 Voters

The voters authenticate to the voting server, select voting options, check the short Choice Return Codes and confirm their vote.

The voters receive the following secret codes by postal mail ([see section 1.1](#)).

Description	Abbreviation
Start Voting Key	SVK
Vector of short Choice Return Codes	CC
Ballot Casting Key	BCK
Vote Cast Return Code	VCC

Tab. 1: Overview of the Voter's codes

2.2 Voting Client

The voting client authenticates to the voting server, encrypts the vote and sends it to the voting server and control components. The voting client works with the following key material.

Abbreviation	Key	Domain
K_{id}, k_{id}	Verification card key pair	$K_{id} \in \mathbb{G}_q, k_{id} \in \mathbb{Z}_q$
EL_{pk}	Election public key	$EL_{pk} \in \mathbb{G}_q^{\delta_{max}}$
pk_{CCR}	Choice Return Codes encryption public key	$pk_{CCR} \in \mathbb{G}_q^{\psi_{max}}$

Tab. 2: List of the Voting Client's keys

2.3 Voting Server

The voting server relays messages to the control components and extracts the short Choice Return Codes and the Vote Cast Return Code.

2.4 Control Components

The control components generate the return codes, shuffle the encrypted votes, and decrypt them at the end of the election event. Conceptually, we distinguish two control component functionalities:

Return Codes control components CCR: compute the Choice Return Codes and the Vote Cast Return Code—in interaction with the setup component (configuration phase) and the voting server (voting phase).

Mixing control components CCM: mix and partially decrypt ciphertexts containing the encrypted votes.

In the specification, we refer to each functionality separately, even though the control components are a single entity combining the CCR and CCM functionalities.

Abbreviation	Key	Domain
$\text{EL}_{\text{pk},j}, \text{EL}_{\text{sk},j}$	CCM _j election key pair	$\text{EL}_{\text{pk},j} \in \mathbb{G}_q^{\delta_{\max}}, \text{EL}_{\text{sk},j} \in \mathbb{Z}_q^{\delta_{\max}}$
EL_{pk}	Remaining election public key	$\text{EL}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\max}}$
K'_j, k'_j	CCR _j Return Codes Generation key pair	$K'_j \in \mathbb{G}_q, k'_j \in \mathbb{Z}_q$
$K_{j,\text{id}}, k_{j,\text{id}}$	Voter Choice Return Code Generation key	$K_{j,\text{id}} \in \mathbb{G}_q, k_{j,\text{id}} \in \mathbb{Z}_q$
$\text{Kc}_{j,\text{id}}, \text{kc}_{j,\text{id}}$	Voter Vote Cast Return Code Generation key	$\text{Kc}_{j,\text{id}} \in \mathbb{G}_q, \text{kc}_{j,\text{id}} \in \mathbb{Z}_q$
$\text{pk}_{\text{CCR}_j}, \text{sk}_{\text{CCR}_j}$	CCR _j Choice Return Codes encryption keys	$\text{pk}_{\text{CCR}_j} \in \mathbb{G}_q^{\psi_{\max}}, \text{sk}_{\text{CCR}_j} \in \mathbb{Z}_q^{\psi_{\max}}$

Tab. 3: List of the Control Component's keys

2.5 Electoral Board

The cantons set up an electoral board comprising citizens or party representatives to observe the orderly processing of an election event. Each member of the electoral board possesses a password, the combination of which derives the electoral board secret key EB_{sk} used for the final decryption of votes. Thereby, we leverage the electoral board as an additional operational safeguard enforcing the four-eyes principle for the final decryption of the votes. The electoral board interacts with the setup component during the configuration phase and with the tally control component during the tally phase.

2.6 Setup Component

The setup component combines the control component's contributions and generates the codes. The setup component is active only during the configuration phase and its software runs in a controlled, offline environment on the canton's premises. Table 4 summarizes the setup component's keys which consist of an ElGamal key pair for encrypting data during the configuration phase.

Abbreviation	Key	Domain
$\text{pk}_{\text{setup}}, \text{sk}_{\text{setup}}$	Setup encryption key pair	$\text{pk}_{\text{setup}} \in \mathbb{G}_q^{n_{\max}}, \text{sk}_{\text{setup}} \in \mathbb{Z}_q^{n_{\max}}$

Tab. 4: List of the Setup Component's keys

2.7 Printing Component

The printing component prints the code sheets and sends them to the voter. All operations in the setup and printing component are subject to strict four-eyes principles and are executed on hardened laptops with special access rights.

2.8 Tally Control Component

The tally control component derives the secret key for the final decryption of the votes from the electoral board members' passwords. Moreover, the Tally Control Component handles the tasks required for the output of the cryptographic protocol: it factorizes the decrypted votes, decodes them to the corresponding voters' selections, and tallies the results into the expected format. Those tasks are fully deterministic and verifiable. The tally control component runs in a controlled, offline environment on the canton's premises.

Abbreviation	Key	Domain
$\text{EB}_{\text{pk}}, \text{EB}_{\text{sk}}$	Electoral board key pair	$\text{EB}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\text{max}}}, \text{EB}_{\text{sk}} \in \mathbb{Z}_q^{\delta_{\text{max}}}$

Tab. 5: List of the Tally Control Component's keys

2.9 Auditors

Auditors verify that the parties faithfully executed their operations within the Swiss Post Voting System. To this end, they employ software as a technical aid: the verifier. We require that at least one trustworthy auditor/verifier checks an election event. The process to select auditors and develop a verifier is outside of the scope of this document.

The verifier does *not* generate keys or participate in calculating the secret codes: we limit the verifier's role to running various verifications.

A verifier specification [21] details the verifier's algorithms.

2.10 Dispute Resolver

The dispute resolver intervenes when control components fail to agree on the list of confirmed votes, a prerequisite for initiating the tally process. As an independent entity, the dispute resolver reconciles these discrepancies by executing the dispute resolution protocol outlined in section 6.2. This process occurs in a controlled, offline environment on cantonal premises.

3 Preliminaries

3.1 Basic Data Types

The [crypto primitives specification](#) details how we represent, convert, and operate on basic data types such as bytes, integers, strings, and arrays.

3.2 Cryptographic Parameters and System Security Level

We refer the reader to the [crypto primitives specification](#) for a detailed description of the cryptographic parameters and the associated security level.

The setup component generates the election event encryption parameters with algorithm 3.1. To prevent modulo overflow when encoding votes, the algorithm ensures that the product of the largest possible combination of primes is smaller than p .

The input `seed` to algorithm 3.1 is restricted to be in the format `CT_YYYYMMDD_XYnm`, where:

- CT: Two uppercase letters for the cantonal abbreviation.
- YYYYMMDD: Date of the election event, e.g. 20231022.
- XYnm: TT/TP/PP, followed by the ascending sequence number, e.g. TT01. The abbreviations have the following meaning:
 - TT: Test event on test environment (TT)
 - TP: Test event on production environment (TP)
 - PP: Productive event on production environment (PP)

Algorithm 3.1 GetElectionEventEncryptionParameters

Context:

Maximum supported number of voting options $n_{\text{sup}} \in \mathbb{N}^+$ ▷ See section 3.5.1
 Maximum supported number of selections $\psi_{\text{sup}} \in \mathbb{N}^+$ ▷ $\psi_{\text{sup}} < n_{\text{sup}}$, see table 10

Input:

`seed` $\in \mathbb{A}_{UCS}^{16}$ ▷ The name of the election event in the format specified above

Operation:

▷ For all algorithms see the crypto primitives specification

- 1: $(p, q, g) \leftarrow \text{GetEncryptionParameters}(\text{seed})$
 - 2: $(p_0, \dots, p_{n_{\text{sup}}-1}) \leftarrow \text{GetSmallPrimeGroupMembers}(p, q, g, n_{\text{sup}})$
 - 3: **if** $\prod_{i=(n_{\text{sup}}-\psi_{\text{sup}})}^{n_{\text{sup}}-1} p_i \geq p$ **then**
 return \perp
 ▷ The product of the ψ_{sup} largest prime numbers cannot be equal to or larger than p
 - 4: **end if**
-

Output:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Small primes $\mathbf{p} = (p_0, \dots, p_{n_{\text{sup}}-1})$, $p_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$, $(p_i < p_{i+1}) \forall i \in [0, n_{\text{sup}})$

3.3 Keys and Codes of the Voting Card

The following table describes the elements of the voting card received by the voters ([see section 1.1](#)).

Description	Abbreviation	Character length	Alphabet	Alphabet Size
Start Voting Key	SVK	$l_{\text{SVK}} = 24$	\mathbb{A}_{u32}	$ \mathbb{A}_{u32} = 32$
Short Choice Return Code	CC	$l_{\text{CC}} = 4$	\mathbb{A}_{10}	$ \mathbb{A}_{10} = 10$
Ballot Casting Key	BCK	$l_{\text{BCK}} = 9$	\mathbb{A}_{10}	$ \mathbb{A}_{10} = 10$
Vote Cast Return Code	VCC	$l_{\text{VCC}} = 8$	\mathbb{A}_{10}	$ \mathbb{A}_{10} = 10$

Tab. 6: Overview of the Voter’s codes. The alphabet \mathbb{A}_{u32} used for the Start Voting Key is described in the crypto primitives specification.

For the authentication of the voter we require a 128-bit security strength, which is considered secure against brute-force attacks. To keep a good balance between usability and security, we use a Start Voting Key with a length of 24 characters in a Base32 alphabet (see table 6), and then apply the Argon2 key derivation function, described in the crypto primitives specification. ~~Percival [17] showed that applying a suitable key derivation function increases the security of the generated key.~~

The length of the Start Voting Key ~~at most provides a security strength of~~ [provides up to](#)

$$e = \lfloor l_{\text{SVK}} \cdot \log_2(|\mathbb{A}_{u32}|) \rfloor = 120 \text{ bits.}$$

~~By applying the Argon2 of entropy. To make brute-force attacks against this space computationally infeasible, we apply the Argon2id key derivation function, we can achieve the desired.~~ As discussed by Percival [17], this raises the cost of offline guessing attacks substantially, compensating for the small entropy shortfall in practice. Argon2id makes exhaustive key search over 120 bits practically infeasible and provides effective resistance comparable to 128-bit security ~~strength~~ in practice. See also RFC 9106 [3, section 7] for security considerations for Argon2id.

In the crypto primitives specification we define different Argon2id profiles. Each time Argon2id is used, we specify which of these profiles is used.

[In contrast to the SVK, the other codes on the voting card, CC, BCK, and VCC, are not required to meet the same entropy standards, as they are not used as cryptographic keys. For usability, they are chosen as short numeric codes. See also the computational proof of the cryptographic protocol \[20, section 19\].](#)

[The codes CC and VCC serve to provide individual verifiability to voters. To protect against guessing attacks, their length ensure that the probability of an attacker correctly guessing a valid code remains below 0.1%, in accordance with the requirements of the Federal Chancellery’s Ordinance \[7\].](#)

[The BCK is entered by the voter to confirm the casting of the ballot, and can only be used within the same session in which the correct corresponding SVK was used. To prevent brute-force attacks, the number of allowed attempts is limited to five per verification card. This restriction makes systematic guessing attacks ineffective.](#)

3.4 Election Event Context

3.4.1 Data Model

We identify each *election event* with the election event ID ee . We group each voter id into a verification card set vcs (usually, these entities correspond to the ensemble of voting cards for a specific municipality) and assign each voter id a verification card vc_{id} and a voting card vcd_{id} . The distinction between *verification card* and *voting card* is merely for technical reasons, and in this document we always refer to the verification card vc_{id} . Moreover, the setup component and the voting client derive a voter's identifier $credentialID_{id}$ from the Start Voting Key SVK_{id} (see section 3.7).

Each verification card set contains $N_{E_{vcs}}$ verification cards. After the voting phase, $N_{C_{bb}}$ voters of a specific verification card set confirmed their vote and thus cast their ballot into the corresponding ballot box bb . A ballot box may contain unconfirmed votes. Since we only tally confirmed votes, the `GetMixnetInitialCiphertexts` algorithm (see section 6.1.1) omits unconfirmed votes. For any verification card set and corresponding ballot box, we have $N_{C_{bb}} \leq N_{E_{vcs}}$. For simplicity, in any algorithm where we do not handle several verification card sets or ballot boxes, we refer to $N_{E_{vcs}}$ simply as N_E and to $N_{C_{bb}}$ simply as N_C . We state $N_C \leq N_E$. Figure 6 summarizes the election event context.

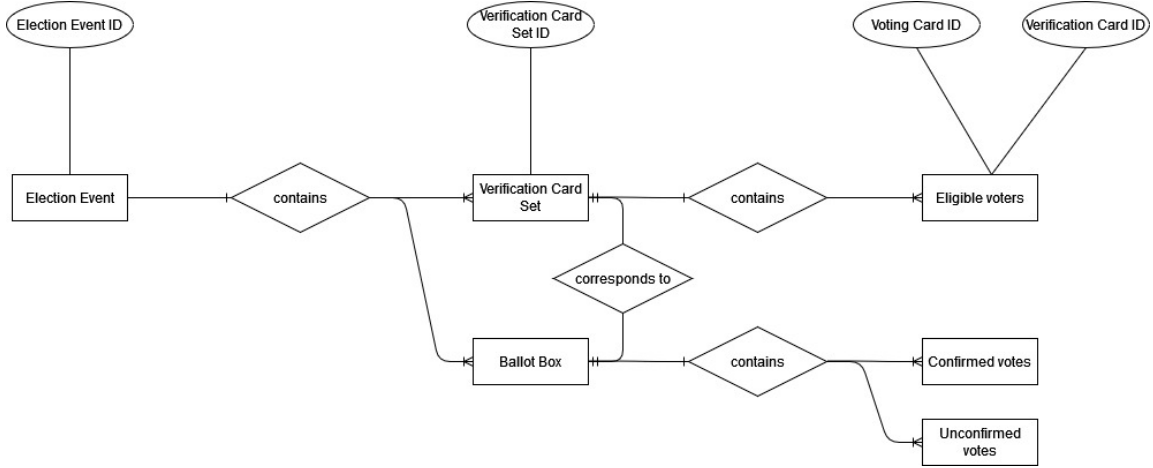


Fig. 6: Overview of the election event context. Every election event has multiple verification card sets. Each verification card set has a corresponding ballot box. A verification card set has N_E eligible voters; a ballot box contains N_C confirmed votes.

We define an election event's vector of verification card set IDs $\mathbf{vcs} = (vcs_0, \dots, vcs_{N_{bb}-1})$ ~~and vector of ballot box IDs $(bb_0, \dots, bb_{N_{bb}-1})$ with the IDs in lexicographic order~~, where N_{bb} is the number of ballot boxes of an election event. ~~The setup component sends the vector of ballot boxes to the control components during the configuration phase.~~

It is important to note that while the voters of the *same* verification card set have the same voting rights, voters of *different* verification card sets may have different ones. For instance, a verification card set could encompass voters of a municipality that has an election on the federal and municipal level while another verification card set contains voters who only vote for the federal election (no municipality-level elections). Therefore, the voting options \hat{v} , the number of selectable voting options ψ , the total number of voting options n , and the number of write-in options $\delta - 1$ can differ across verification card sets (see section 3.5).

The election authorities can also define *test* verification card sets/ballot boxes to validate the correct configuration of the election event prior to the actual voting and tally phase. Furthermore, we assume that the election authorities define the ballot boxes in a way that guarantees vote secrecy; if a ballot box contains only one vote, this voter's privacy is trivially broken.

3.4.2 Hashing the Election Event Context

~~We define an algorithm `GetHashElectionEventContext` that hashes the global election event context. This algorithm is used in the Configuration phase by the algorithms `SetupTallyCCM` and `SetupTallyEB` to ensure consensus of the Election Event Context between the Control Components and the Setup Component.~~

~~The start and finish times of the election event and of the ballot boxes are restricted to be in local date time format (YYYY-MM-DDTHH:MM:SS) without time zone as defined by the ISO-8601 calendar system. The time zone used will always be the local time zone in Switzerland, therefore no indication of timezone is necessary.~~

~~`GetHashElectionEventContext`~~

~~Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$~~

~~Election event ID $ee \in (\mathbb{A}_{Base16})^{1ID}$ Election Event alias $eeAlias \in \mathcal{T}_1^{50}$ Election Event~~

~~description $eeDesc \in \mathbb{A}_{UCS}^*$ Verification Card Set Contexts Election Event start time~~

~~$t_{s,ee} \in \mathbb{A}_{UCS}^{19}$ Election Event finish time $t_{f,ee} \in \mathbb{A}_{UCS}^{19}$ Maximum number of voting options~~

~~$n_{max} \in [1, n_{sup}]$ Maximum number of selections $\psi_{max} \in [1, \psi_{sup}]$ Maximum number of write-ins~~

~~$+1$: $\delta_{max} \in [1, \delta_{sup}]$~~

~~Verification Card Set Contexts: For each of the N_{bb} verification card sets $vcs \in \mathbf{vcs}$~~

~~Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1ID}$ Verification card set alias $vcsAlias \in \mathcal{T}_1^{50}$~~

~~Verification card set description $vcsDesc \in \mathbb{A}_{UCS}^*$ Ballot box ID $bb \in (\mathbb{A}_{Base16})^{1ID}$ Ballot box~~

~~start time $t_{s,bb} \in \mathbb{A}_{UCS}^{19}$ Ballot box finish time $t_{f,bb} \in \mathbb{A}_{UCS}^{19}$ Test ballot box boolean~~

~~$testBallotBox \in \{"true", "false"\}$ Number of eligible voters for this verification card set~~

~~$N_E \in \mathbb{N}^+$ Grace period $gracePeriod \in [0, 3600]$~~

~~$h_{pTable,j} \leftarrow \left((p, q, g), \left((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}) \right) \right)$~~

~~$h_{vcs,j} \leftarrow (vcs, vcsAlias, vcsDesc, bb, t_{s,bb}, t_{f,bb}, testBallotBox, N_E, gracePeriod, h_{pTable,j})$~~

~~$h_{vcs} \leftarrow (h_{vcs,0}, h_{vcs,1}, \dots, h_{vcs, N_{bb}-1})$ $h \leftarrow (ee, eeAlias, eeDesc, h_{vcs}, t_{s,ee}, t_{f,ee}, n_{max}, \psi_{max}, \delta_{max})$~~

~~$d \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h))$~~

~~The digest $d \in \mathbb{A}_{Base64}^{1HR64}$~~

~~Test values for the algorithm 3.11 are provided in `get-hash-election-event-context.json`.~~

3.4.2 Execution Context of Algorithms

Each algorithm executes within a given context; sometimes, the protocol runs an algorithm per election event, per verification card set, per ballot box, or actual vote. Table 7 summarizes the algorithms' execution context.

Algorithm	Execution context	Reference
GenSetupData	per election event	4.1
GenVerDat	per verification card set	4.2
GetVoterAuthenticationData	per verification card set	4.3
GenKeysCCR	per election event	4.4
GenEncLongCodeShares	per verification card set	4.5
CombineEncLongCodeShares	per verification card set	4.6
GenCMTable	per verification card set	4.9
GenVerCardSetKeys	per election event	4.10
GenCredDat	per verification card set	4.11
SetupTallyCCM	per election event	4.12
SetupTallyEB	per election event	4.13
Voting phase algorithms	per sent/confirmed vote	section 5
Tally phase <u>MixOnline</u> algorithms	per ballot box	section 6.1 <u>6.1</u>
<u>Dispute resolution algorithms</u>	<u>per election event</u>	<u>section 6.2</u>
<u>VerifyVotingClientProofs</u>	<u>per ballot box</u>	<u>6.9</u>
<u>VerifyMixDecOffline</u>	<u>per ballot box</u>	<u>6.10</u>
<u>MixDecOffline</u>	<u>per ballot box</u>	<u>6.11</u>
<u>ProcessPlaintexts</u>	<u>per ballot box</u>	<u>6.12</u>
<u>CreateECH0222</u>	<u>per election event</u>	<u>section 6.3.5</u>
<u>RequestProofNonParticipation</u>	<u>per verification card</u>	<u>6.13</u>

Tab. 7: Overview of the algorithms' context. The protocol executes some configuration phase algorithms globally per election event and some per verification card set. The voting phase algorithms run for every sent or confirmed vote, ~~while all~~. Most tally phase algorithms ~~run~~ are executed per ballot box. In contrast, the dispute resolution process and the generation of the tally file are performed globally for the entire election event. The proof of non-participation executes individually for a specific verification card.

3.5 Electoral Model

3.5.1 Electoral Model and Parameters

The Swiss Post Voting System supports the electoral model outlined in the eCH-standards. An eCH standard is a guideline developed by the eCH association to ensure interoperability and quality in Swiss e-government projects and processes. The eCH-0155 standard [10], in combination with the eCH-0045 standard on electoral registers [9], defines the data format for attributes related to votes and elections at all political levels in Switzerland.

The electoral model described in the eCH-0155 standard boils down to support $t \geq 1$ simultaneous, independent ψ_j -out-of- n_j elections, where the voter selects for every election $j \in [0, \dots, t)$ exactly ψ_j voting options out of n_j possible ($\psi_j \leq n_j$).

Across all t elections, $\psi = \sum_{j=0}^{t-1} \psi_j$ represents the cumulative number of selections, while $n = \sum_{j=0}^{t-1} n_j$ signifies the aggregate number of voting options.

A similar approach was taken by the authors of the CHVote system [13].

The total number of voting options, n_j in an election is determined by several factors: the number of candidates (broadly defined to include lists and potential answers to posed questions), permitted accumulations, and the number of blank and write-in voting options. Accumulation, specifying the repeat count for candidate nominations on a ballot, ranges from 1 to 3, with the higher values reserved for proportional elections. If the accumulation is set to 2 or 3, the system assigns 2 or 3 voting options to this specific candidate; each voting option having a different Choice Return Code (see section 1.1). The number of blank voting options equates to the number of selections, ψ_j , ensuring an equal count of blank options to selections. The number of write-in options, either zero or equal to the selection count, are contingent on their allowance in a given election, with their use exclusive to majoral elections.

ψ_j is a parameter of the election, whereas n_j is derived from multiplying the number of candidates by the accumulation value, and adding the totals of blank voting options and write-in options. In elections where both a list and multiple individual candidates are selected, we break down the election into two distinct sub-elections: one for the list selection and another for the candidate selection.

The election type for questions can be seen as a special case of a majoral election (with 2 candidates and 1 seat) and the electoral model could be easily adapted to accommodate for questions with more than the 3 standard answers (YES / NO / BLANK).

Table 8 highlights examples of ψ_j and n_j for common supported electoral models.

Election Type	Number of Candidates	Accumulation	Blank Voting Options	Write-in Voting Options	ψ_j	n_j
Question (e.g. initiative, referendum)	2 (YES and NO)	1	1	0	1	3
Majoral election with 5 candidates and 2 seats without write-ins	5	1	2	0	2	7
Majoral election with 5 candidates and 2 seats with write-ins	5	1	2	2	2	9
Proportional election (list part) with 12 lists	12	1	1	0	1	13
Proportional election (candidates part) with 180 candidates, 30 seats, and accumulation 2	180	2	30	0	30	390

Tab. 8: Example parameters of the electoral model.

Table 9 defines the symbols for the election event parameters configured for a specific verification card set.

Parameters	Actual value (configured per verification card set)
Number of voting options	n
Selectable voting options	ψ
Number of write-in options	$\delta - 1$

Tab. 9: Verification card set specific parameters.

Hence, a voter must select exactly ψ out of n voting options. We derive the verification card set specific values n , ψ , and δ from the primes mapping table **pTable** (see section 3.5.3). Alongside the specific parameters for each verification card set, the algorithm **GenSetupData** (see section 4.1.1) derives these global parameters for the entire *election event*. Table 10 shows these parameters as well as the maximum value supported by the system for these settings, as per our analysis of federal, cantonal, and municipal elections.

Parameters	Actual value (global per election event)	Maximum supported value
Maximum number of voting options	n_{\max}	$n_{\sup} = 5000$
Maximum number of selections	ψ_{\max}	$\psi_{\sup} = \text{120150}$
Maximum number of write-ins	$\delta_{\max} - 1$	$\delta_{\sup} - 1 = 30$

Tab. 10: Election event specific parameters.

Algorithm 4.13 generates the election public key EL_{pk} with exactly δ_{max} elements.

3.5.2 Encoding of Voting Options

We denote the actual voting options as a vector of strings $\tilde{\mathbf{v}} \leftarrow (v_0, \dots, v_{n-1})$, $v_i \in \mathcal{T}_1^{50}$.

Sometimes, the canton's configuration of the election event only guarantees that the identifiers of the actual voting options are unique for a specific election, but not across different elections. Therefore, we define the actual voting options as follows:

- Lists: election identifier | list identifier
- Candidates: election identifier | candidate identifier | candidate accumulation
- Blank candidate position: election identifier | blank candidate position identifier
- Write-in position: election identifier | write-in position identifier
- Answer: question identifier | answer identifier

For simplicity, we will throughout this document always refer to the domain of the actual voting options as \mathcal{T}_1^{50} . In practice, each of the identifiers above is represented with a string from \mathcal{T}_1^{50} , and we separate them using the pipe character |. Thus, the domain of an actual voting option is ~~either \mathcal{T}_1^{50} | \mathcal{T}_1^{50} or~~

~~\mathcal{T}_1^{50} | \mathcal{T}_1^{50} | \mathcal{T}_1^{50} | $\{S_1 \parallel " " \parallel S_2 : S_1, S_2 \in \mathcal{T}_1^{50}\} \cup \{S_1 \parallel " " \parallel S_2 \parallel " " \parallel S_3 : S_1, S_2, S_3 \in \mathcal{T}_1^{50}\}$.~~

The voting options are encoded as prime numbers $\tilde{\mathbf{p}} \leftarrow (\tilde{p}_0, \dots, \tilde{p}_{n-1})$, $\tilde{p}_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$ (maintaining the order between both vectors). ~~The cryptographic primitives~~

~~specification [19] details the algorithms for generating the~~

Algorithm 3.1 GetElectionEventEncryptionParameters details the generation of small primes used to encode voting options. The mapping of voting options to prime numbers is *bijective*: each voting option maps to a distinct prime number and vice versa.

Moreover, we denote the semantic information corresponding to each voting option as a vector of strings $\sigma \leftarrow (\sigma_0, \dots, \sigma_{n-1})$, $\sigma_i \in \mathbb{A}_{UCS}^*$. We assume that the election event configuration contains German, French, Italian, and Rumantsch. We construct the voting option's semantic information σ_i in the following way:

- Non-blank list: "NON_BLANK" | list description (de, fr, it, rm)
- Blank list: "BLANK" | list description (de, fr, it, rm)
- Candidate: "NON_BLANK" | family name | ~~first name~~ | call name | date of birth
- Blank candidate position: "BLANK" | EMPTY_CANDIDATE_POSITION-[position number]
- Write-in position: "WRITE_IN" | WRITE_IN_POSITION-[position number]
- Non-blank answer: "NON_BLANK" | question (de, fr, it, rm) | answer (de, fr, it, rm)
- Blank answer: "BLANK" | question (de, fr, it, rm) | answer (de, fr, it, rm)

Finally, every voting option belongs either to a question, to a list selection of an election, or to a candidate selection of an election. Since this information is very important to assess whether a vote is *correct*, i.e. whether it conforms to a predefined way of filling out the ballot, we assign to each actual voting option v_i a correctness information τ_i .

We construct the correctness information τ_i in the following way:

- If v_i corresponds to an answer of a referendum-style question, use the question identifier as τ_i .
- If v_i corresponds to a list in an election (or to the empty list), use the prefix “L” concatenated with the pipe character (|) and the election identifier as τ_i .
- If v_i corresponds to a candidate, empty or write-in position in an election, use the prefix “C” concatenated with the pipe character (|) and the election identifier as τ_i .

This results in the vector of correctness information $\boldsymbol{\tau} = (\tau_0, \dots, \tau_{n-1})$.

3.5.3 Primes Mapping Table

We define a primes mapping table **pTable**, conceptually, as the combination of $\tilde{\mathbf{v}}$, $\tilde{\mathbf{p}}$, $\boldsymbol{\sigma}$, and $\boldsymbol{\tau}$ and represented as an ordered list of tuples, *i.e.* $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$. ~~If an election event has multiple elections and votes, the~~ The primes mapping table must sequentially organize voting options as they appear in the election event configuration: ~~voting options for the initial question come first, followed by~~. We first order the voting options by votePosition and electionGroupPosition. Then, we sort within the election groups and votes. Within an election group, the voting options are sorted by electionPosition. Within a single election, the voting options ~~for the second question, etc. Similarly, lists and candidates for the first election come first, followed by the lists and candidates for the second election.~~ are sorted in the following way:

1. If the election has lists, the list entries, sorted by listOrderOfPrecedence.
2. If the election has lists, the empty list entry.
3. The candidate entries. If the election has lists, then in order of appearance, otherwise sorted by candidatePosition.
4. The blank position entries, in order of appearance.
5. If the election has write-ins, the write-in position entries, in order of appearance.

Within a single ~~election or question, no specific arrangement of voting options is necessary~~ vote, the voting options are sorted by ballotPosition. Within a standard ballot the voting options are sorted by answerPosition. Within a variant ballot the standard and tie-break questions are sorted by questionPosition, and within each question the voting options are sorted by answerPosition.

The setup component generates the primes mapping table **pTable** in the algorithm 4.1 **GenSetupData** and sends it to the auditors (see figure 7) and to the Tally control component (see figure 8). The cryptographic protocol ensures that all participants have the same view of **pTable** by linking it to the Verification Card Keystore \mathbf{VCks}_{id} and the voting client’s zero-knowledge proofs. In particular, the following algorithms ensure that all protocol participants have a consistent view of **pTable**:

- The setup component includes the contents of **pTable** in the authenticated symmetric encryption in 4.11 **GenCredDat**.

- The voting client includes the contents of **pTable** in the authenticated symmetric decryption in 5.3 **GetKey** and in the zero-knowledge proofs in 5.4 **CreateVote**.
- The control components include the contents of **pTable** when verifying the voting client's zero-knowledge proofs in 5.5 **VerifyBallotCCR**.
- The control components include the contents of **pTable** when generating a zero-knowledge proof in 5.6 **PartialDecryptPCC** and verifying each other's proofs in 5.7 **DecryptPCC**.
- The tally control component includes the contents of **pTable** when verifying the voting client's zero-knowledge proofs in 6.9 **VerifyVotingClientProofs**.
- The auditors include the contents of **pTable** when verifying the voting client's zero-knowledge proofs in **VerifyTally**.

The following algorithms operate on the primes mapping table **pTable**. As a special case, algorithms 3.2, 3.3, and 3.5 can be called with an empty list argument to retrieve all corresponding elements of the **pTable**.

Algorithm 3.2 GetEncodedVotingOptions

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Input:

Actual voting options $(v'_0, \dots, v'_{m'-1}) \in (\mathcal{T}_1^{50})^{m'}$

Require: $0 \leq m' \leq n$

Require: $v'_i \in \mathbf{pTable} \forall i \in \{0, \dots, m' - 1\}$

Require: $v'_i \neq v'_k, \forall i, k \in \{0, \dots, m' - 1\} \wedge i \neq k \triangleright$ All actual voting options must be distinct

Operation:

```

1: if  $m' = 0$  then
2:    $m \leftarrow n$ 
3:    $(\tilde{p}'_0, \dots, \tilde{p}'_{m-1}) \leftarrow (\tilde{p}_0, \dots, \tilde{p}_{m-1})$ 
4: else
5:    $m \leftarrow m'$ 
6:   for  $i \in [0, m)$  do
7:      $\tilde{p}'_i \leftarrow \mathbf{pTable}(v'_i)$   $\triangleright$  Retrieve the encoded voting option from pTable
8:   end for
9: end if

```

Output:

Encoded voting options $(\tilde{p}'_0, \dots, \tilde{p}'_{m-1}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^m$

Algorithm 3.3 GetActualVotingOptions

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Input:

Encoded voting options $(\tilde{p}'_0, \dots, \tilde{p}'_{m'-1}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{m'}$

Require: $0 \leq m' \leq n$

Require: $\tilde{p}'_i \in \mathbf{pTable} \forall i \in \{0, \dots, m' - 1\}$

Require: $\tilde{p}'_i \neq \tilde{p}'_k, \forall i, k \in \{0, \dots, m' - 1\} \wedge i \neq k \quad \triangleright$ All encoded voting options must be distinct

Operation:

```

1: if  $m' = 0$  then
2:    $m \leftarrow n$ 
3:    $(v'_0, \dots, v'_{m-1}) \leftarrow (v_0, \dots, v_{m-1})$ 
4: else
5:    $m \leftarrow m'$ 
6:   for  $i \in [0, m)$  do
7:      $v'_i \leftarrow \mathbf{pTable}(\tilde{p}'_i) \quad \triangleright$  Retrieve the actual voting option from  $\mathbf{pTable}$ 
8:   end for
9: end if

```

Output:

Actual voting options $(v'_0, \dots, v'_{m-1}) \in (\mathcal{T}_1^{50})^m$

Algorithm 3.4 GetSemanticInformation

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Operation:

```

1: return  $(\sigma_0, \dots, \sigma_{n-1})$ 

```

Output:

Semantic information $(\sigma_0, \dots, \sigma_{n-1}) \in (\mathbb{A}_{UCS}^*)^n$

Algorithm 3.5 GetCorrectnessInformation

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Input:

Actual voting options $(v'_0, \dots, v'_{m'-1}) \in (\mathcal{T}_1^{50})^{m'}$

Require: $0 \leq m' \leq n$

Require: $v'_i \in \mathbf{pTable} \forall i \in \{0, \dots, m' - 1\}$

Require: $v'_i \neq v'_k, \forall i, k \in \{0, \dots, m' - 1\} \wedge i \neq k \triangleright$ All actual voting options must be distinct

Operation:

```

1: if  $m' = 0$  then
2:    $m \leftarrow n$ 
3:    $(\tau'_0, \dots, \tau'_{m-1}) \leftarrow (\tau_0, \dots, \tau_{m-1})$ 
4: else
5:    $m \leftarrow m'$ 
6:   for  $i \in [0, m)$  do
7:      $\tau'_i \leftarrow \mathbf{pTable}(v'_i)$   $\triangleright$  Retrieve the correctness information from  $\mathbf{pTable}$ 
8:   end for
9: end if

```

Output:

Correctness information $(\tau'_0, \dots, \tau'_{m-1}) \in (\mathcal{T}_1^{50})^m$

Furthermore, we define additional algorithms that allow us to leverage the structured information contained in the voting options' semantic information.

First, the algorithm **GetBlankCorrectnessInformation** returns the vector of correctness information corresponding to the blank voting options. Since our electoral model (see section 3.5.1) ensures that there are as many blank voting options as selections, the resulting vector $\hat{\tau}$ is of size ψ . Section 3.5.4 explains how we leverage this information to ensure that a vote contains a valid combination of voting options.

Algorithm 3.6 GetBlankCorrectnessInformation

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Operation:

```

1:  $k \leftarrow 0$ 
2: for  $i \in [0, n)$  do
3:   if  $\sigma_i$  starts with “BLANK” then
4:      $\hat{\tau}_k \leftarrow \mathbf{pTable}(\sigma_i)$   $\triangleright$  Retrieve the correctness information  $\tau_i$  from  $\mathbf{pTable}$ 
5:      $k \leftarrow k + 1$ 
6:   end if
7: end for
8: if  $k = 0$  then
9:   return  $\perp$   $\triangleright$  There must be at least one blank voting option
10: end if
11: return  $(\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1})$   $\triangleright$  The resulting vector must have the size  $\psi$  since it corresponds to a
    valid vote with all blank voting options.

```

Output:

Blank correctness information $(\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1}) \in (\mathcal{T}_1^{50})^\psi$

Second, we define a method **GetWriteInEncodedVotingOptions** that returns the encoded voting options that correspond to write-in voting options.

Algorithm 3.7 GetWriteInEncodedVotingOptions

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Operation:

```

1:  $k \leftarrow 0$ 
2: for  $i \in [0, n)$  do
3:   if  $\sigma_i$  starts with “WRITE_IN” then
4:      $\tilde{p}_{w,k} \leftarrow \mathbf{pTable}(\sigma_i)$   $\triangleright$  Retrieve the encoded voting option  $\tilde{p}_i$  from  $\mathbf{pTable}$ 
5:      $k \leftarrow k + 1$ 
6:   end if
7: end for
8: return  $(\tilde{p}_{w,0}, \dots, \tilde{p}_{w,\delta-2})$   $\triangleright$  The resulting vector has the size  $\delta - 1 \geq 0$ 

```

Output:

Encoded write-in voting options $(\tilde{p}_{w,0}, \dots, \tilde{p}_{w,\delta-2}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{\delta-1}$

Moreover, we define two auxiliary algorithms **GetPsi** and **GetDelta** calculating the number of selectable voting options ψ and the number of write-in options + 1: δ .

Algorithm 3.8 GetPsi

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Operation:

- 1: $(\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1}) \leftarrow \text{GetBlankCorrectnessInformation}()$ \triangleright See algorithm 3.6
 - 2: **return** ψ
-

Output:

The number of selectable voting options: $\psi \in [1, \psi_{\text{sup}}]$

Algorithm 3.9 GetDelta

Context:

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Operation:

- 1: $(\tilde{p}_{w,0}, \dots, \tilde{p}_{w,\delta-2}) \leftarrow \text{GetWriteInEncodedVotingOptions}()$ \triangleright See algorithm 3.7
 - 2: **return** δ
-

Output:

The number of allowed write-ins + 1: $\delta \in [1, \delta_{\text{sup}}]$

3.5.4 Valid Combinations of Voting Options

The Federal Chancellery's Ordinance [7] requires that an e-voting system must not accept invalid votes.

[VEleS Annex 10]: Conformity check and storing finalised votes:
- Votes not cast in conformity with the system
are not stored in the electronic ballot box.

The Ordinance's annex further elaborates on the definition of *votes cast in conformity with the system*.

[VEleS Annex 2.1]: *vote cast in conformity with the system* is a vote:
- that complies with a predetermined way of completing a ballot
paper in a vote or election
- [...].

In the electoral model outlined in Section 3.5.1, the essence is that a vote must encompass ψ selections and adhere to a valid combination of voting options. Specifically, this means each vote must include the appropriate number of selections for each election or question. For instance, if the ballot has two questions—each with the corresponding voting options YES / NO / EMPTY—we expect the voter to select exactly *one* voting option for the first question and *one* for the second question. Conversely, if a voter were to select an invalid combination, for instance, YES and NO to the same question, her vote would be considered invalid.

To prevent invalid combinations of voting options, we utilize the principle detailed in section 3.5.1, which specifies an equivalence between the number of blank voting options and selections. Moreover, we can leverage the fact that each component of the Swiss Post Voting System knows the primes mapping table **pTable**. The primes mapping table **pTable** associates each voting option with a correctness information τ_i ; resulting in a vector $\boldsymbol{\tau} = (\tau_0, \dots, \tau_{n-1})$. The configuration of the election event also defines how many voting options from which vote or election can be selected and in which order; resulting in a vector of correct selections $\hat{\boldsymbol{\tau}} = (\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1})$. We can derive the vector $\hat{\boldsymbol{\tau}}$ by invoking the algorithm **GetBlankCorrectnessInformation** on the primes mapping table **pTable**.

We illustrate the aforementioned concept with a simple example. Imagine the following election event configuration:

- One question with three voting options (YES / NO / EMPTY) - question identifier = "question₁",
- One question with three voting options (YES / NO / EMPTY) - question identifier = "question₂",
- An election (without lists) for three seats and five candidates - election identifier = "election₁".

Such an election event configuration indicates that the total number of selectable voting options ψ is 5 (one for the first question, one for the second, and three for the election), and if no accumulation of candidates is allowed, the total number of voting options n is 14 (three for

the first question, three for the second, and eight for the election, including three blank seat selections). Invoking the algorithm **GetBlankCorrectnessInformation** for this election event configuration returns the following vector

(“question₁”, “question₂”, “C|election₁”, “C|election₁”, “C|election₁”).

The Swiss Post Voting System checks that a vote contains a valid combination of voting options at several stages within the protocol:

- Within the voting client, during the creation of the vote, as outlined in algorithm 5.4 **CreateVote**.
- Within the control components during the execution of the **SendVote** protocol, specifically through algorithm 5.8 **CreateLCCShare**. Here, the control components incorporate the correctness information $\hat{\tau}$ into the evaluation of the partial Choice Return Codes allow list L_{pcc} , initially generated by the setup component through algorithm 4.2 **GenVerDat**.
- Within both the verifier and the tally control component, following the decryption of votes, as detailed in algorithm 6.12 **ProcessPlaintexts**.

While our electoral model treats each election as independent, ensuring valid combinations of voting options presents challenges, particularly in proportional elections where voters select both a party list and candidates. In the case of the national council election, a vote for a party list without at least one non-blank candidate selection is deemed invalid. However, the use of correctness information $\hat{\tau}$ does not address this issue, as invoking

GetBlankCorrectnessInformation for a party list without candidate selections would still yield the expected correctness vector, failing to identify the vote as invalid.

To mitigate this, the Swiss Post Voting System currently employs user interface controls within the voting client to prevent the submission of such invalid combinations. Additionally, during the generation of ~~tally file~~the eCH-0222 XML, as discussed in Section ~~??~~6.3.5, any invalid vote is filtered out to ensure it is not counted.

Future versions of the Swiss Post Voting System will improve the detection and prevention of this type of invalid combination as part of the ongoing improvements outlined in Measure A.26 of the Federal Chancellery’s catalogue of measures [6].

3.5.5 Hashing the Voting Options' Context

We define an algorithm `GetHashContext` that hashes the cryptographic parameters (section 3.2), the verification card set specific context (section 3.4), the primes mapping table `pTable` as well as the election public key `ELpk` and the Choice Return Codes encryption public key `pkCCR`.

`GetHashContext`

Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$
Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$ Verification card set ID $ves \in (\mathbb{A}_{Base16})^{1_{ID}}$ Election public key $EL_{pk} = (EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$ Choice Return Codes encryption public key $pk_{CCR} = (pk_{CCR,0}, \dots, pk_{CCR,\psi_{max}-1}) \in \mathbb{G}_q^{\psi_{max}}$
 $h \leftarrow ()$ $h \leftarrow (h, \text{"EncryptionParameters"}, p, q, g)$ $h \leftarrow (h, \text{"ElectionEventContext"}, ee, ves)$
 $h \leftarrow (h, \text{"ActualVotingOptions"}, \text{GetActualVotingOptions}())$
 $h \leftarrow (h, \text{"EncodedVotingOptions"}, \text{GetEncodedVotingOptions}())$
 $h \leftarrow (h, \text{"SemanticInformation"}, \text{GetSemanticInformation}())$
 $h \leftarrow (h, \text{"CorrectnessInformation"}, \text{GetCorrectnessInformation}())$
 $h \leftarrow (h, \text{"ELpk"}, EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1})$ $h \leftarrow (h, \text{"pkCCR"}, pk_{CCR,0}, \dots, pk_{CCR,\psi_{max}-1})$
 $d \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h))$
The digest $d \in \mathbb{A}_{Base64}^{1_{HR64}}$
Test values for the algorithm 3.12 are provided in `get-hash-context.json`.

3.5.5 Multiplying and Factorizing Voting Options

The voter `id` selects ψ unique encoded voting options $\hat{\mathbf{p}}_{id} = (\hat{p}_{id,0}, \dots, \hat{p}_{id,\psi-1})$. The ψ -element vector of selected, encoded voting options $\hat{\mathbf{p}}_{id}$ is a subset of the n -element vector of encoded voting options $\tilde{\mathbf{p}}$ ($\hat{\mathbf{p}}_{id} \subset \tilde{\mathbf{p}}$). We verify in algorithm 3.1 that the product of any subset of ψ_{sup} primes (the system's maximum supported number of selections - see section 3.5.1) is smaller than the group modulus p to ensure that all voting combinations are uniquely represented. To avoid encrypting multiple messages, we multiply the encoded voting options prior to encryption (see the algorithm 5.4 `CreateVote`):

$$\rho = \prod_{i=0}^{\psi-1} \hat{p}_{id,i} \in \mathbb{G}_q$$

Conversely, we factorize the encoded voting options after decryption (see the algorithm 6.12 `ProcessPlaintexts`):

$$\hat{\mathbf{p}}_{id} = \text{Factorize}(\rho)$$

Factorizing the message is efficient since the primes are small and the set of primes is known. Since every voting option is selected at most once and each voter must select ψ voting options, we expect *exactly* ψ distinct prime factors.

Algorithm 3.10 Factorize

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Encoded voting options $\tilde{\mathbf{p}} = (\tilde{p}_0, \dots, \tilde{p}_{n-1}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^n \triangleright$ Can be derived from **pTable** using algorithm 3.2

Number of allowed selections for this specific ballot box $\psi \in [1, \psi_{\text{sup}}] \triangleright$ Can be derived from **pTable** using algorithm 3.8

Input:

$x \in \mathbb{G}_q$

\triangleright The number to be factorized

Operation:

1: $i \leftarrow 0$

2: **for** $k \in [0, n)$ **do**

3: **if** $\tilde{p}_k \mid x$ **then**

4: $\hat{p}_i \leftarrow \tilde{p}_k$

5: $i = i + 1$

6: **end if**

7: **end for**

8: **if** $i \neq \psi \vee \prod_{j=0}^{\psi-1} \hat{p}_j \neq x$ **then**

return $\perp \triangleright$ The factorization is only successful if the number of prime factors is ψ and their product equals x .

9: **end if**

Output:

$\hat{\mathbf{p}} = (\hat{p}_0, \dots, \hat{p}_{\psi-1}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^\psi$

\triangleright The list of prime factors

3.6 Agreement Algorithms

3.6.1 Agreement on the Global Election Event Context

At specific stages of the cryptographic protocol, the Swiss Post Voting System requires the protocol parties to agree on the election event context (see section 3.4) and the applicable electoral model (see section 3.5).

Therefore, we define an algorithm `GetHashElectionEventContext` that hashes the global election event context. This algorithm is used in the configuration phase by the `Setup Component` in algorithm `SetupTallyEB` and by the `Control Components` in algorithm `SetupTallyCCM`, ensuring consensus of the Election Event Context between these components.

The start and finish times of the election event and of the ballot boxes are restricted to be in local date time format (YYYY-MM-DDTHH:MM:SS) without time zone as defined by the ISO-8601 calendar system. The time zone used will always be the local time zone in Switzerland, therefore no indication of timezone is necessary.

Algorithm 3.11 GetHashElectionEventContext

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
Election Event alias $eeAlias \in \mathcal{T}_1^{50}$
Election Event description $eeDesc \in \mathbb{A}_{UCS}^*$
Verification Card Set Contexts ▷ See below for the contents
Election Event start time $t_{s,ee} \in \mathbb{A}_{UCS}^{19}$ ▷ Start and finish time in the format stated above
Election Event finish time $t_{f,ee} \in \mathbb{A}_{UCS}^{19}$
Maximum number of voting options $n_{max} \in [1, n_{sup}]$
Maximum number of selections $\psi_{max} \in [1, \psi_{sup}]$
Maximum number of write-ins + 1: $\delta_{max} \in [1, \delta_{sup}]$
Verification Card Set Contexts: For each of the N_{bb} verification card sets $vcs \in \mathbf{vcs}$ ▷ Given in lexicographic order
Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1_{ID}}$
Verification card set alias $vcsAlias \in \mathcal{T}_1^{50}$
Verification card set description $vcsDesc \in \mathbb{A}_{UCS}^*$
Ballot box ID $bb \in (\mathbb{A}_{Base16})^{1_{ID}}$
Ballot box start time $t_{s,bb} \in \mathbb{A}_{UCS}^{19}$ ▷ Start and finish time in the format stated above
Ballot box finish time $t_{f,bb} \in \mathbb{A}_{UCS}^{19}$
Test ballot box boolean $testBallotBox \in \{\text{"true"}, \text{"false"}\}$
Number of eligible voters for this verification card set $N_E \in \mathbb{N}^+$
Grace period $gracePeriod \in [0, 3600]$ ▷ The grace period is given in seconds
Primes mapping table $pTable \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ ▷ $pTable$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$
Vector of domains of influence $DoI \in (\mathcal{T}_1^{50})^k$

Operation:

▷ We use nested structures in this algorithm

- 1: **for** $vcs \in \mathbf{vcs}$ **do**
- 2: $h_{pTable,j} \leftarrow \left(\left((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}) \right) \right)$
- 3: $h_{vcs,j} \leftarrow (vcs, vcsAlias, vcsDesc, bb, t_{s,bb}, t_{f,bb}, testBallotBox, N_E, gracePeriod, h_{pTable,j}, DoI)$
- 4: **end for**
- 5: $h_{vcs} \leftarrow (h_{vcs,0}, h_{vcs,1}, \dots, h_{vcs, N_{bb}-1})$
- 6: $h \leftarrow ((p, q, g), ee, eeAlias, eeDesc, h_{vcs}, t_{s,ee}, t_{f,ee}, n_{max}, \psi_{max}, \delta_{max})$
- 7: $d \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h))$ ▷ See crypto primitives specification

Output:

The digest $d \in \mathbb{A}_{Base64}^{1_{HB64}}$ ▷ 1_{HB64} is the character length of the Base64 encoded hash output

Test values for the algorithm 3.11 are provided in [get-hash-election-event-context.json](#).

3.6.2 Agreement on the Verification Card Set-Specific Data

At specific stages of the cryptographic protocol, the Swiss Post Voting System requires the protocol parties to agree on the verification card set-specific context, which includes elements of the election event context as well as certain cryptographic data and public keys.

Therefore, we define an algorithm `GetHashContext` that hashes the cryptographic parameters (section 3.2), the verification card set specific context (section 3.4), the primes mapping table `pTable` as well as the election public key `ELpk` and the Choice Return Codes encryption public key `pkCCR`.

This algorithm is used in the configuration phase by the Setup Component and in the voting phase by the Voting Client and the Control Components.

Algorithm 3.12 `GetHashContext`

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1_{ID}}$

Primes mapping table $pTable \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright pTable$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Election public key $EL_{pk} = (EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$

Choice Return Codes encryption public key $pk_{CCR} = (pk_{CCR,0}, \dots, pk_{CCR,\psi_{max}-1}) \in \mathbb{G}_q^{\psi_{max}}$

Operation:

\triangleright We flatten the lists in this algorithm

- 1: $h \leftarrow ()$
 - 2: $h \leftarrow (h, \text{"EncryptionParameters"}, p, q, g)$
 - 3: $h \leftarrow (h, \text{"ElectionEventContext"}, ee, vcs)$
 - 4: $h \leftarrow (h, \text{"ActualVotingOptions"}, \text{GetActualVotingOptions}())$ \triangleright See 3.3
 - 5: $h \leftarrow (h, \text{"EncodedVotingOptions"}, \text{GetEncodedVotingOptions}())$ \triangleright See 3.2
 - 6: $h \leftarrow (h, \text{"SemanticInformation"}, \text{GetSemanticInformation}())$ \triangleright See 3.4
 - 7: $h \leftarrow (h, \text{"CorrectnessInformation"}, \text{GetCorrectnessInformation}())$ \triangleright See 3.5
 - 8: $h \leftarrow (h, \text{"ELpk"}, EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1})$
 - 9: $h \leftarrow (h, \text{"pkCCR"}, pk_{CCR,0}, \dots, pk_{CCR,\psi_{max}-1})$
 - 10: $d \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h))$ \triangleright See crypto primitives specification
-

Output:

The digest $d \in \mathbb{A}_{Base64}^{1_{HB64}}$ $\triangleright 1_{HB64}$ is the character length of the Base64 encoded hash output

Test values for the algorithm 3.12 are provided in [get-hash-context.json](#).

3.6.3 Agreement on the Entire Election Event-Specific Data

At specific stages of the cryptographic protocol, the Swiss Post Voting System requires the protocol parties to agree on the entire election event-specific data, including both the election event context (Section 3.6.1) and the verification card set-specific data (Section 3.6.2). Therefore, we define an algorithm `GetHashExtractedElectionEvent` that hashes the entire election event-specific data. `GetHashExtractedElectionEvent` is used by the Control Components in the voting phase in algorithms `PartialDecryptPCC` and `DecryptPCC`. To facilitate the algorithm `GetHashExtractedElectionEvent`, we define the algorithm `ExtractElectionEvent`, which extracts all relevant information required to represent the entirety of the election event, with helper algorithm `ExtractVerificationCardSet`.

We assume the existence of functions [ExtractElectionEventContext](#), [ExtractElectionPublicKey](#), and [ExtractChoiceReturnCodesEncryptionPublicKey](#) that retrieve the necessary values from internal view.

Algorithm 3.13 ExtractElectionEvent

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Operation:

▷ All extraction algorithms retrieve the necessary values from internal view

- 1: $electionEventContext \leftarrow \text{ExtractElectionEventContext}(ee)$
- 2: $hContext \leftarrow \text{GetHashElectionEventContext}()$ ▷ See algorithm 3.11
- 3: $EL_{pk} \leftarrow \text{ExtractElectionPublicKey}(ee)$
- 4: $pk_{CCR} \leftarrow \text{ExtractChoiceReturnCodesEncryptionPublicKey}(ee)$
- 5: **for** $i \in [0, N_{bb})$ **do**
- 6: $evcs_i \leftarrow \text{ExtractVerificationCardSet}()$ ▷ See algorithm 3.14
- 7: **end for**
- 8: $evcs \leftarrow (evcs_0, \dots, evcs_{N_{bb}-1})$
- 9: $evcs \leftarrow \text{Order}(evcs)$ ▷ Lexicographic ordering by the verification card set id.

Output:

Extracted election event $eee = (hContext, (p, q, g), ee, evcs)$

▷ $evcs = (h, vcs, L_{pcc}, L_{lvcc})$

We assume the existence of functions [ExtractPartialChoiceReturnCodesAllowList](#) and [ExtractLongVoteCastReturnCodesAllowList](#) that retrieve the necessary values from internal view.

Algorithm 3.14 ExtractVerificationCardSet

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1_{ID}}$

Primes mapping table $pTable \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ ▷ $pTable$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$

Election public key $EL_{pk} = (EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$

Choice Return Codes encryption public key $pk_{CCR} = (pk_{CCR,0}, \dots, pk_{CCR,\psi_{max}-1}) \in \mathbb{G}_q^{\psi_{max}}$

Operation:

▷ All extraction algorithms retrieve the necessary values from internal view

- 1: $h \leftarrow \text{GetHashContext}()$ ▷ See algorithm 3.12
- 2: $L_{pcc} \leftarrow \text{ExtractPartialChoiceReturnCodesAllowList}(vcs)$
- 3: $L_{lvcc} \leftarrow \text{ExtractLongVoteCastReturnCodesAllowList}(vcs)$

Output:

Extracted verification card set $evcs = (h, vcs, L_{pcc}, L_{lvcc})$

Finally, we define the algorithm GetHashExtractedElectionEvent that hashes the extracted election event $\text{eee} = (\text{hContext}, (p, q, g), \text{ee}, \text{evcs})$.

Algorithm 3.15 GetHashExtractedElectionEvent

Context:

Extracted election event $\text{eee} = (\text{hContext}, (p, q, g), \text{ee}, \text{evcs})$

$\triangleright \text{evcs} = (\text{h}, \text{vcs}, \text{L}_{\text{pcc}}, \text{L}_{\text{ivcc}})$

Operation:

\triangleright We use nested structures in this algorithm

- 1: **for** $i \in [0, N_{\text{bb}})$ **do**
- 2: $\text{h}_{\text{evcs}, i} \leftarrow (\text{h}, \text{vcs}, \text{L}_{\text{pcc}}, \text{L}_{\text{ivcc}})$
- 3: **end for**
- 4: $\text{h}_{\text{evcs}} \leftarrow (\text{h}_{\text{evcs}, 0}, \dots, \text{h}_{\text{evcs}, N_{\text{bb}}-1})$
- 5: $h \leftarrow (\text{hContext}, (p, q, g), \text{ee}, \text{h}_{\text{evcs}})$
- 6: $d \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h))$

\triangleright See crypto primitives specification

Output:

The digest $d \in \mathbb{A}_{\text{Base64}}^{\text{l}_{\text{HB64}}}$
output

$\triangleright \text{l}_{\text{HB64}}$ is the character length of the Base64 encoded hash

3.6.4 Agreement on the Sent Votes from the Voting Phase

We describe the control components' `ExtractVerificationCards` algorithm. This algorithm is only invoked in the dispute resolution process (see section 6.2). As a *precondition*, the `ExtractVerificationCards` algorithm can only be invoked after the election event period ended. We assume the existence of functions `ExtractVerificationCardSetId`, `ExtractEncryptedVote`, and `ExtractHashedLVCCShares` that retrieve the necessary values from internal view.

Algorithm 3.16 `ExtractVerificationCards`

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Stateful Lists and Maps:

CCR_j's list of sent voting cards $L_{sentVotes,j}$

CCR_j's list of confirmed voting cards $L_{confirmedVotes,j}$

Operation:

- ▷ All extraction algorithms retrieve the necessary values from internal view
- ▷ We use nested structures in this algorithm

```

1: for  $vc_{id} \in L_{sentVotes,j}$  do
2:    $vcs \leftarrow \text{ExtractVerificationCardSetId}(vc_{id})$ 
3:    $E1 \leftarrow \text{ExtractEncryptedVote}(vc_{id})$ 
4:   if  $vc_{id} \in L_{confirmedVotes,j}$  then
5:      $(hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}) \leftarrow \text{ExtractHashedLVCCShares}(vc_{id})$ 
6:      $evc \leftarrow (vc_{id}, vcs, E1, (hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}))$ 
7:   else
8:      $evc \leftarrow (vc_{id}, vcs, E1, ())$ 
9:   end if
10: end for
11:  $evc \leftarrow (evc_0, \dots, evc_{N_s-1})$ 
12:  $evc \leftarrow \text{Order}(evc)$ 

```

▷ Lexicographic ordering by the verification card id.

Output:

Extracted verification cards $evc = (evc_0, \dots, evc_{N_s-1})$

3.6.5 Proof of Correct Key Generation

In the algorithms `GenKeysCCR`, `SetupTallyCCM`, and `SetupTallyEB` the control components and the setup component generate shares of ElGamal encryption key pairs, subsequently distributing the public keys among the other protocol participants. Bernhard et al. showed that this approach might be vulnerable to an attack [2]: a malicious party k could delay its key pair generation until after receiving the public keys from honest participants. This party could then generate a key pair (pk_k, sk_k) and send a rogue public key $pk'_k = \prod_{i \neq k} \frac{pk_k}{pk_i}$ to the other participants. The resulting combined public key $pk = \prod_{i \neq k} pk_i \cdot pk'_k$ would then correspond to the attacker's public key pk_k .

To prevent this attack, each participant creates a Schnorr proof of knowledge of their secret key matching the distributed public key.

Since multiple protocol participants verify these proofs throughout the protocol, the following section specifies the algorithms for verifying the Schnorr proofs of knowledge for correct key generation.

If the public keys are available from multiple sources—for instance from a message and from an internal view—one must check their consistency when invoking the algorithm `VerifyKeyGenerationSchnorrProofs`.

Algorithm 3.17 VerifyKeyGenerationSchnorrProofs

Context:

The Election Event Context

▷ See section 3.4

▷ Including $p, q, g, ee, \psi_{\max}, \delta_{\max}$

Input:

CCR Choice Return Codes encryption public keys

$\mathbf{pk}_{\text{CCR}} = (\mathbf{pk}_{\text{CCR}_1}, \mathbf{pk}_{\text{CCR}_2}, \mathbf{pk}_{\text{CCR}_3}, \mathbf{pk}_{\text{CCR}_4}) \in (\mathbb{G}_q^{\psi_{\max}})^4$

CCR Schnorr proofs of knowledge $\pi_{\mathbf{pk}_{\text{CCR}}} = (\pi_{\mathbf{pk}_{\text{CCR},1}}, \pi_{\mathbf{pk}_{\text{CCR},2}}, \pi_{\mathbf{pk}_{\text{CCR},3}}, \pi_{\mathbf{pk}_{\text{CCR},4}}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi_{\max}})^4$

CCM election public keys $\mathbf{EL}_{\mathbf{pk}} = (\mathbf{EL}_{\mathbf{pk},1}, \mathbf{EL}_{\mathbf{pk},2}, \mathbf{EL}_{\mathbf{pk},3}, \mathbf{EL}_{\mathbf{pk},4}) \in (\mathbb{G}_q^{\delta_{\max}})^4$

CCM Schnorr proofs of knowledge $\pi_{\mathbf{EL}_{\mathbf{pk}}} = (\pi_{\mathbf{EL}_{\mathbf{pk},1}}, \pi_{\mathbf{EL}_{\mathbf{pk},2}}, \pi_{\mathbf{EL}_{\mathbf{pk},3}}, \pi_{\mathbf{EL}_{\mathbf{pk},4}}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}})^4$

Electoral board public key $\mathbf{EB}_{\mathbf{pk}} \in \mathbb{G}_q^{\delta_{\max}}$

Electoral board Schnorr proofs of knowledge $\pi_{\mathbf{EB}} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}}$

Operation:

- 1: $\mathbf{hContext} \leftarrow \text{GetHashElectionEventContext}()$ ▷ See algorithm 3.11
 - 2: $\text{VerifSchnorrCCR} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{pk}_{\text{CCR}}, \pi_{\mathbf{pk}_{\text{CCR}}}, (ee, \text{“GenKeysCCR”}))$ ▷ See algorithm 3.18
 - 3: $\text{VerifSchnorrCCM} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{EL}_{\mathbf{pk}}, \pi_{\mathbf{EL}_{\mathbf{pk}}}, (\mathbf{hContext}, \text{“SetupTallyCCM”}))$
 - 4: $\text{VerifSchnorrEB} \leftarrow \text{VerifyCCSchnorrProofs}((\mathbf{EB}_{\mathbf{pk}}), (\pi_{\mathbf{EB}}), (\mathbf{hContext}, \text{“SetupTallyEB”}))$
 - 5: **if** $\text{VerifSchnorrCCR} \wedge \text{VerifSchnorrCCM} \wedge \text{VerifSchnorrEB}$ **then**
 - 6: **return** \top
 - 7: **else**
 - 8: **return** \perp
 - 9: **end if**
-

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

Algorithm 3.18 VerifyCCSchnorrProofs

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Number of components to verify $N_v \in \{1, 4\}$
Number of elements of the public key $N_{pk} \in \mathbb{N}^+$

Input:

Vector of public keys $\mathbf{pk}_{CC} \in (\mathbb{G}_g^{N_{pk}})^{N_v}$
Vector of Schnorr proofs of knowledge $\pi_{\mathbf{pk}_{CC}} \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{N_{pk}})^{N_v}$
The additional information $\mathbf{i}_{aux} \in (\mathbb{A}_{UCS}^*)^s, s \in \mathbb{N}$

Operation: \triangleright For all algorithms see the crypto primitives specification \triangleright We flatten the lists in this algorithm

```

1: for  $j \in [1, N_v]$  do
2:    $\mathbf{i}_{aux,j} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(j))$ 
3:   for  $i \in [0, N_{pk})$  do
4:      $\text{VerifSchnorrCC}_{j,i} \leftarrow \text{VerifySchnorr}(\pi_{\mathbf{pk}_{CC},j,i}, \mathbf{pk}_{CC,j,i}, \mathbf{i}_{aux,j})$ 
5:   end for
6: end for
7: if  $\text{VerifSchnorrCC}_{j,i} \forall j, i$  then
8:   return  $\top$ 
9: else
10:  return  $\perp$ 
11: end if

```

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

3.7 Voter Authentication

Authentication of the voter to the voting server requires the derivation of a unique voter identifier and a base authentication challenge by both the setup component and the voting client

To this end, the `DeriveCredentialId` algorithm generates fresh identifiers for each election event by instantiating `Argon2id` with a “salt” containing the election event ID and a hard-coded string. Although it is recommended that `Argon2` uses a unique salt for each password (see [3]), in our case, we use the same salt for all voters participating in a given election event. This does not compromise security since the setup component generates a unique high-entropy password for each voter, the Start Voting Key SVK_{id} , which is used by the voting client to invoke `Argon2`. Furthermore, when the algorithm 3.19 is first invoked, the voting client has not yet received any unique information about the voter that could be used as a salt for the `Argon2` algorithm.

Algorithm 3.19 `DeriveCredentialId`

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Input:

Start Voting Key $SVK_{id} \in (\mathbb{A}_{u32})^{1_{SVK}}$

Operation:

▷ For all algorithms see the crypto primitives specification

- 1: $salt \leftarrow \text{CutToBitLength}(\text{RecursiveHash}(ee, \text{“credentialId”}), 128)$
 - 2: $bcredentialID_{id} \leftarrow \text{GetArgon2id}(\text{StringToByteArray}(SVK_{id}), salt)$ ▷ Use the Argon2 profile for less memory, as defined in the crypto primitives specification
 - 3: $credentialID_{id} \leftarrow \text{Base16Encode}(\text{CutToBitLength}(bcredentialID_{id}, 128))$
-

Output:

$credentialID_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Furthermore, both the setup component and the voting client need to generate a base authentication challenge using their shared secrets: the Start Voting Key and the extended authentication factor. Algorithm 3.20 is used to create this base authentication challenge. The extended authentication factor is explained in 4.1.3.

We use the same salt per election event for all voters for the same reasons as in algorithm 3.19.

Algorithm 3.20 DeriveBaseAuthenticationChallenge

Context:

Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Character length of the extended authentication factor $1_{EA} \in \mathbb{N}^+$

Input:

Start Voting Key $\mathbf{SVK}_{id} \in (\mathbb{A}_{u32})^{1_{SVK}}$

Extended authentication factor $\mathbf{EA}_{id} \in \mathbb{A}_{10}^{1_{EA}}$

Operation:

▷ For all algorithms see the crypto primitives specification

- 1: $\mathbf{salt}_{auth} \leftarrow \text{CutToBitLength}(\text{RecursiveHash}(\mathbf{ee}, \text{"hAuth"}), 128)$
 - 2: $\mathbf{k} \leftarrow (\text{StringToByteArray}(\mathbf{EA}_{id}) \parallel \text{StringToByteArray}(\text{"Auth"}) \parallel \text{StringToByteArray}(\mathbf{SVK}_{id}))$
 - 3: $\mathbf{bhAuth}_{id} \leftarrow \text{GetArgon2id}(\mathbf{k}, \mathbf{salt}_{auth})$ ▷ Use the Argon2 profile for less memory, as defined in the crypto primitives specification
 - 4: $\mathbf{hAuth}_{id} \leftarrow \text{Base64Encode}(\mathbf{bhAuth}_{id})$
-

Output:

Base authentication challenge $\mathbf{hAuth}_{id} \in (\mathbb{A}_{Base64})^{1_{HB64}}$

3.8 Write-Ins

Certain elections in Switzerland allow voters to write a candidate's name in a form instead of selecting a predefined candidate (so-called write-in candidates). Usually, only a tiny fraction of voters select write-in candidates.

For write-in candidates, the Federal Chancellery's Ordinance waives the requirement of individual verifiability [7].

[VEleS art. 16 para. 2]: The Federal Chancellery may in exceptional cases exempt a canton from meeting individual requirements, provided:

- a. the requirements that have not been met are indicated in the application;
- b. reasonable justification is brought forward for allowing an exemption; and
- c. the canton describes any alternative measures and justifies in reference to the risk assessment why it regards the risks as sufficiently low.

The explanatory report elaborates on that exemption [8].

[Explanatory Report, Sec 4.2.1]: Para. 2: In exceptional cases, a canton may be exempted from meeting individual requirements. This option is subject to the three conditions set out on letters a-c. In particular, there must be a clear justification for making an exception. An exception might be: in an election run using the first-past-the-post system, the requirement of individual verifiability can be waived if the vote is cast by entering a name in blank text field ('write-in votes')

Moreover, the explanatory report clarifies that write-in votes are always *cast in conformity with the system*.

[Explanatory Report, Sec 4.2.1]: Let. o No 1: In elections run using the first-past-the-post system, blank text fields ('write-in votes') are always considered to have been completed in conformity with the system.

The Swiss Post Voting System supports the usage of write-in candidates along predefined candidates. However, the security guarantees regarding individual verifiability apply only to predefined voting options.

Using multi-recipient ElGamal encryption, the system encodes and encrypts write-in candidates as separate messages along with the predefined candidates (which are still encoded as the product of primes and linked to a specific Choice Return Code using NIZK proofs).

The following two algorithms are mainly affected by the presence of write-in candidates:

- 5.4 CreateVote
- 6.12 ProcessPlaintexts

The voter must choose to enter a write-in candidate for a given position and enters a write-in from a given alphabet. The voting client will encode the write-in to a group element and encrypts the encoded write-in as part of the multi-recipient ElGamal ciphertext.

Upon decryption of the ciphertext votes in the Tally control component, the **ProcessPlaintexts** decodes the ciphertext elements back to the write-in contents.

3.8.1 Alphabet used for Write-Ins

For the write-ins, we use the alphabet $\mathbb{A}_{\text{latin}}$ described in the crypto primitives specification. The size of this alphabet is $|\mathbb{A}_{\text{latin}}| = a$. The symbol $\#$, which is the rank 0 character in the alphabet $\mathbb{A}_{\text{latin}}$, is by the system used as a separator and must therefore be prevented within the write-in fields. A trustworthy voting client will prevent usage of $\#$ within the write-in fields, as seen in algorithm 5.4.

3.8.2 Encoding Write-Ins

Encoding a write-in field uses algorithms 3.21 and 3.22:

Algorithm 3.21 WritelnToQuadraticResidue: Map a character string to a value $\in \mathbb{G}_q$

Context:

- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$

Input:

- A character string $s \in \mathbb{A}_{\text{latin}}^*$ ▷ See section 3.8.1

Require: $a^{|s|} < q$ ▷ where a is the size of $\mathbb{A}_{\text{latin}}$ and $|s|$ is the character length of s

Require: $|s| > 0$

Require: s does not start with the rank 0 character (leading 0s are lost by the encoding)

Operation:

- $x \leftarrow \text{WritelnToInteger}(s)$ ▷ See algorithm 3.22
 - $y \leftarrow x^2 \bmod p$
-

Output:

- $y \in \mathbb{G}_q$

Test values for the algorithm 3.21 are provided in [write-in-to-quadratic-residue.json](#).

Algorithm 3.22 WriteInteger: Map a character string to an integer value

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$

Input:

A character string $s \in \mathbb{A}_{\text{latin}}^*$ ▷ See section 3.8.1

Require: $a^{|s|} < q$ ▷ where a is the size of $\mathbb{A}_{\text{latin}}$ and $|s|$ is the character length of s

Require: $|s| > 0$

Require: s does not start with the rank 0 character (leading 0s are lost by the encoding)

Operation:

```

1:  $x \leftarrow 0$ 
2: for  $i \in [0, |s|)$  do ▷  $|s|$  is the character length of  $s$ 
3:    $c \leftarrow$  the character at the  $i^{\text{th}}$  position of  $s$ 
4:    $b \leftarrow$  the rank of character  $c$  within  $\mathbb{A}_{\text{latin}}$  ▷ See section 3.8.1
5:    $x \leftarrow xa + b$  ▷ Leading characters of rank 0 would be ignored.
6: end for
    
```

Output:

$x \in \mathbb{Z}_q \notin \{0\}$ ▷ Since $|s| > 0$ and $a^{|s|} < q$

Test values for the algorithm 3.22 are provided in [write-in-to-integer.json](#).

3.8.3 Decoding Write-Ins

Decoding a write-in field uses algorithms 3.23 and 3.24:

Algorithm 3.23 QuadraticResidueToWriteIn: Map a quadratic residue to a write-in string

Input:

The quadratic residue $y \in \mathbb{G}_q$ ▷ By contract, we assume that y has the correct mathematical group.

Operation:

```

1:  $x \leftarrow y^{(p+1)/4} \bmod p$  ▷ Thus  $x^2 \equiv y^{(p+1)/2} \equiv y^{(2q+2)/2} \equiv y^q \cdot y \equiv y \pmod{p}$ 
2: if  $x > q$  then ▷ We want the smallest of the two roots of  $y$ 
3:    $x \leftarrow p - x$ 
4: end if
5:  $s \leftarrow \text{IntegerToWriteIn}(x)$  ▷ See algorithm 3.24

```

Output:

The string $s \in \mathbb{A}_{\text{latin}}^*$ ▷ See section 3.8.1

Test values for the algorithm 3.23 are provided in [qr-to-write-in.json](#).

Algorithm 3.24 IntegerToWriteIn: Map an integer $\in \mathbb{Z}_q$ to a write-in string

Input:

The encoding $x \in \mathbb{Z}_q \notin \{0\}$ ▷ By contract, we assume that x has the correct mathematical group.

Operation:

```

1:  $s \leftarrow ""$ 
2: while  $x > 0$  do
3:    $b \leftarrow x \bmod a$ 
4:    $c \leftarrow$  the character at rank  $b$  within alphabet  $\mathbb{A}_{\text{latin}}$  ▷ See section 3.8.1
5:    $s \leftarrow c || s$  ▷ String concatenation
6:    $x \leftarrow \frac{(x-b)}{a}$ 
7: end while

```

Output:

The string $s \in \mathbb{A}_{\text{latin}}^*$ ▷ See section 3.8.1

Test values for the algorithm 3.24 are provided in [integer-to-write-in.json](#).

3.8.4 Creating a Vote with Write-Ins

For example, let us assume that we have an election for three seats, where write-ins are allowed. Hence δ is 4 (number of write-in positions + 1). We assume that the voter makes the following choice:

- Position 1: Write-in Candidate “Jane Doe”
- Position 2: Pre-Defined Candidate “Richard Doe”
- Position 3: Write-in Candidate “John Doe”

Here, the voter selected 2 out of 3 possible write-in options. In general, we can say that the voter selects k out of $(\delta - 1)$ write-in options.

In case the voter selected a write-in candidate, the system is going to display to the voter a Choice Return Code indicating that the write-in field was selected.

The algorithm **EncodeWriteIns** converts the write-in entries into a vector of \mathbb{G}_q elements. Since the voter might select less than $(\delta - 1)$ write-ins, **EncodeWriteIns** encodes the write-in contents first and then fills the remaining (unused) write-in positions with dummy values. The algorithm **EncodeWriteIns** requires that the input elements are ordered carefully: the selected write-in candidates must be arranged according to the order of the elections as defined by the election event configuration. This specific order is crucial for accurately attributing write-ins during the tally, particularly when multiple elections permit write-in candidates.

Algorithm 3.25 EncodeWriteIns

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\text{sup}}] \triangleright$ Can be derived from **pTable** using algorithm 3.9

Input:

The vector of selected write-ins $\hat{\mathbf{s}} = (\hat{s}_0, \dots, \hat{s}_{k-1}) \in (\mathbb{A}_{\text{latin}}^*)^k \triangleright$ See section 3.8.1 \triangleright the k write-in contents ordered in the way described in 3.8.4

Require: $k \leq \delta - 1 \quad \triangleright \delta = 1$, if the ballot box does not have any write-in candidates.

Operation:

- 1: **for** $i \in [0, k)$ **do**
 - 2: $w_i \leftarrow \text{WriteInToQuadraticResidue}(\hat{s}_i)$ \triangleright See Algorithm 3.21
 - 3: **end for**
 - 4: **for** $i \in [k, (\delta - 1))$ **do**
 - 5: $w_i \leftarrow 1$ \triangleright Fill the remaining position with dummy values
 - 6: **end for**
-

Output:

Encoded write-ins $\mathbf{w} = (w_0, \dots, w_{\delta-2}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{\delta-1}$

Test values for the algorithm 3.25 are provided in [encode-write-ins.json](#).

3.8.5 Processing Write-Ins in the Tally Phase

Upon decryption, the `ProcessPlaintexts` algorithm decodes the write-ins. The `ProcessPlaintexts` algorithm does *not* decode more write-ins than the number of selected write-in options. We define a method `IsWriteInOption` that determines if a voting option corresponds to a write-in.

Algorithm 3.26 `IsWriteInOption`

Context:

Encoded write-in voting options $\tilde{\mathbf{p}}_w = (\tilde{p}_{w,0}, \dots, \tilde{p}_{w,\delta-2}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{\delta-1} \triangleright$ Can be derived from `pTable` using algorithm 3.7

Input:

Encoded voting option $\tilde{p}_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g \triangleright$ By contract, we assume that \tilde{p}_i has the correct mathematical group.

Operation:

```

1: if  $\tilde{p}_i \in \tilde{\mathbf{p}}_w$  then
    return  $\top$ 
2: else
    return  $\perp$ 
3: end if

```

Output:

\top if the voting option is a write-in, \perp otherwise.

Test values for the algorithm 3.26 are provided in [is-write-in-option.json](#).

To decode write-ins, we thus define an algorithm `DecodeWriteIns` that takes as input a list of encoded voting options and a list of encoded write-ins. Note that `DecodeWriteIns` returns an empty list if the input list of encoded write-ins is empty. Moreover, the **eCH-files** eCH-0222 XML (see section ~~??~~ [6.3.5](#)) ~~impose~~ [imposes](#) certain restrictions on the character length of the write-in contents. The voter portal restricts the character length of the write-in to a much smaller value. However, a malicious voting client could potentially encode a write-in of larger character length. Therefore, the algorithm 3.27 truncates the write-in values to prevent the creation of ~~invalid eCH-files~~ [an invalid](#) eCH-0222 XML.

Algorithm 3.27 DecodeWriteIns

Context:

Encoded write-in voting options $\tilde{\mathbf{p}}_{\mathbf{w}} = (\tilde{p}_{\mathbf{w},0}, \dots, \tilde{p}_{\mathbf{w},\delta-2}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{\delta-1}$ \triangleright Can be derived from **pTable** using algorithm 3.7

Number of allowed selections for this specific ballot box $\psi \in [1, \psi_{\text{sup}}]$ \triangleright Can be derived from **pTable** using algorithm 3.8

Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\text{sup}}]$ \triangleright Can be derived from **pTable** using algorithm 3.9

Input: \triangleright By contract, we assume that the input elements have the correct mathematical group.

Selected encoded voting options $\hat{\mathbf{p}} = (\hat{p}_0, \dots, \hat{p}_{\psi-1}) \in ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^\psi$

List of encoded write-ins $\mathbf{w} = (w_0, \dots, w_{\delta-2}) \in \mathbb{G}_q^{(\delta-1)}$

Operation:

```

1:  $\hat{\mathbf{s}} \leftarrow ()$ 
2: if  $\delta = 1$  then  $\triangleright$  The election event has no write-ins
3:   return  $\hat{\mathbf{s}}$ 
4: end if
5:  $k \leftarrow 0$ 
6: for  $i \in [0, \psi)$  do
7:   if  $\text{IsWriteInOption}(\hat{p}_i)$  then  $\triangleright$  See Algorithm 3.26
8:      $\hat{s}_k \leftarrow \text{Truncate}(\text{QuadraticResidueToWriteIn}(w_k), 1_{\mathbf{w}})$   $\triangleright$  See Algorithm 3.23 and crypto primitives specification
9:      $\hat{\mathbf{s}} \leftarrow (\hat{\mathbf{s}}, \hat{s}_k)$ 
10:     $k \leftarrow k + 1$ 
11:   end if
12: end for

```

Output:

The vector of decoded write-ins $\hat{\mathbf{s}} \in (\mathbb{A}_{\text{latin}}^*)^k$ \triangleright See section 3.8.1 $\triangleright k$ is the number of selected write-in candidates

[Test values for the algorithm 3.27 are provided in `decode-write-ins.json`.](#)

Given the example from section 3.8.4, we would end up with the following results:

- Write-in 1: $\text{Truncate}(\text{QuadraticResidueToWriteIn}(w_{\text{id},0}), 1_{\mathbf{w}})$
- Write-in 2: $\text{Truncate}(\text{QuadraticResidueToWriteIn}(w_{\text{id},1}), 1_{\mathbf{w}})$
- Write-in 3: Nothing to decode, since only two write-in voting options were selected.

3.9 Proof of Correct Key Generation

In the algorithms ~~GenKeysCCR~~, ~~SetupTallyCCM~~, and ~~SetupTallyEB~~ the control components and the setup component generate shares of ElGamal encryption key pairs, subsequently distributing the public keys among the other protocol participants. Bernhard et al. showed that this approach might be vulnerable to an attack [2]: a malicious party k could delay its key pair generation until after receiving the public keys from honest participants. This party could then generate a key pair (pk_k, sk_k) and send a rogue public key $pk'_k = \frac{pk_k}{\prod_{i \neq k} pk_i}$ to the other participants. The resulting combined public key $pk = \prod_{i \neq k} pk_i \cdot pk'_k$ would then correspond to the attacker's public key pk_k .

To prevent this attack, each participant creates a Schnorr proof of knowledge of their secret key matching the distributed public key.

Since multiple protocol participants verify these proofs throughout the protocol, the following section specifies the algorithms for verifying the Schnorr proofs of knowledge for correct key generation.

If the public keys are available from multiple sources—for instance from a message and from an internal view—one must check their consistency when invoking the algorithm

~~VerifyKeyGenerationSchnorrProofs~~.

~~VerifyKeyGenerationSchnorrProofs~~ The Election Event Context

~~CCR~~ Choice Return Codes encryption public keys

$\mathbf{pk}_{CCR} = (\mathbf{pk}_{CCR,1}, \mathbf{pk}_{CCR,2}, \mathbf{pk}_{CCR,3}, \mathbf{pk}_{CCR,4}) \in (\mathbb{G}_q^{\psi_{\max}})^4$ CCR Schnorr proofs of knowledge

$\pi_{\mathbf{pk}_{CCR}} = (\pi_{\mathbf{pk}_{CCR},1}, \pi_{\mathbf{pk}_{CCR},2}, \pi_{\mathbf{pk}_{CCR},3}, \pi_{\mathbf{pk}_{CCR},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi_{\max}})^4$ CCM election public keys

$\mathbf{EL}_{\mathbf{pk}} = (\mathbf{EL}_{\mathbf{pk},1}, \mathbf{EL}_{\mathbf{pk},2}, \mathbf{EL}_{\mathbf{pk},3}, \mathbf{EL}_{\mathbf{pk},4}) \in (\mathbb{G}_q^{\delta_{\max}})^4$ CCM Schnorr proofs of knowledge

$\pi_{\mathbf{EL}_{\mathbf{pk}}} = (\pi_{\mathbf{EL}_{\mathbf{pk}},1}, \pi_{\mathbf{EL}_{\mathbf{pk}},2}, \pi_{\mathbf{EL}_{\mathbf{pk}},3}, \pi_{\mathbf{EL}_{\mathbf{pk}},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}})^4$ Electoral board public key

$\mathbf{EB}_{\mathbf{pk}} \in \mathbb{G}_q^{\delta_{\max}}$ Electoral board Schnorr proofs of knowledge $\pi_{\mathbf{EB}} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}}$

$\mathbf{hContext} \leftarrow \text{GetHashElectionEventContext}()$

$\text{VerifSchnorrCCR} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{pk}_{CCR}, \pi_{\mathbf{pk}_{CCR}}, (\mathbf{ee}, \text{"GenKeysCCR"}))$

$\text{VerifSchnorrCCM} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{EL}_{\mathbf{pk}}, \pi_{\mathbf{EL}_{\mathbf{pk}}}, (\mathbf{hContext}, \text{"SetupTallyCCM"}))$

$\text{VerifSchnorrEB} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{EB}_{\mathbf{pk}}, \pi_{\mathbf{EB}}, (\mathbf{hContext}, \text{"SetupTallyEB"})) \top \perp$

The result of the verification: \top if the verification is successful, \perp otherwise.

~~VerifyCCSchnorrProofs~~ Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$ Number of components to verify $N_V \in \{1, 4\}$ Number of elements of the public key $N_{\mathbf{pk}} \in \mathbb{N}^{\pm}$

Vector of public keys $\mathbf{pk}_{CC} \in (\mathbb{G}_q^{N_{\mathbf{pk}}})^{N_V}$ Vector of Schnorr proofs of knowledge

$\pi_{\mathbf{pk}_{CC}} \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{N_{\mathbf{pk}}})^{N_V}$ The additional information $\mathbf{i}_{\text{aux}} \in (\mathbb{A}_{UCS}^*)^s, s \in \mathbb{N}$

$\mathbf{i}_{\text{aux},j} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(j))$

$i \in [0, N_{\mathbf{pk}}) \text{VerifSchnorrCC}_{j,i} \leftarrow \text{VerifySchnorr}(\pi_{\mathbf{pk}_{CC},j,i}, \mathbf{pk}_{CC,j,i}, \mathbf{i}_{\text{aux},j}) \top \perp$

The result of the verification: \top if the verification is successful, \perp otherwise.

4 Configuration Phase

The configuration phase consists of two sub-protocols: ~~SetupVoting and SetupTally~~SetupVoting and SetupTally. Table 11 highlights the algorithms involved.

Algorithm	Actor	Reference
SetupVoting		Figure 7
GenSetupData	Setup Component	4.1
GenVerDat	Setup Component	4.2
GetVoterAuthenticationData	Setup Component	4.3
GenKeysCCR	Control Component (CCR)	4.4
GenEncLongCodeShares	Control Component (CCR)	4.5
CombineEncLongCodeShares	Setup Component	4.6
GenCMTable	Setup Component	4.9
GenVerCardSetKeys	Setup Component	4.10
GenCredDat	Setup Component	4.11
SetupTally		Figure 8
SetupTallyCCM	Control Component (CCM)	4.12
SetupTallyEB	Setup Component	4.13
VerifyConfigPhase	Auditors	Verifier Specification [21]

Tab. 11: Overview of the algorithms in the configuration phase.

~~Moreover, we assume that~~ Section 4.3 elaborates on the data flows and agreement mechanisms at the end of the configuration phase, ~~all components have received the Election Event Context. See sections 3.4 and 3.5~~ as well as the transition of the election event's configuration to the voting phase.

The control components ensure that the algorithms listed in table 11 can only be executed during the configuration phase, and not after it has ended. The setup component is only used in the configuration phase, and remains unused and offline throughout the voting and tally phases.

4.1 SetupVoting

Figure 7 depicts the ~~SetupVoting~~SetupVoting sub-protocol.

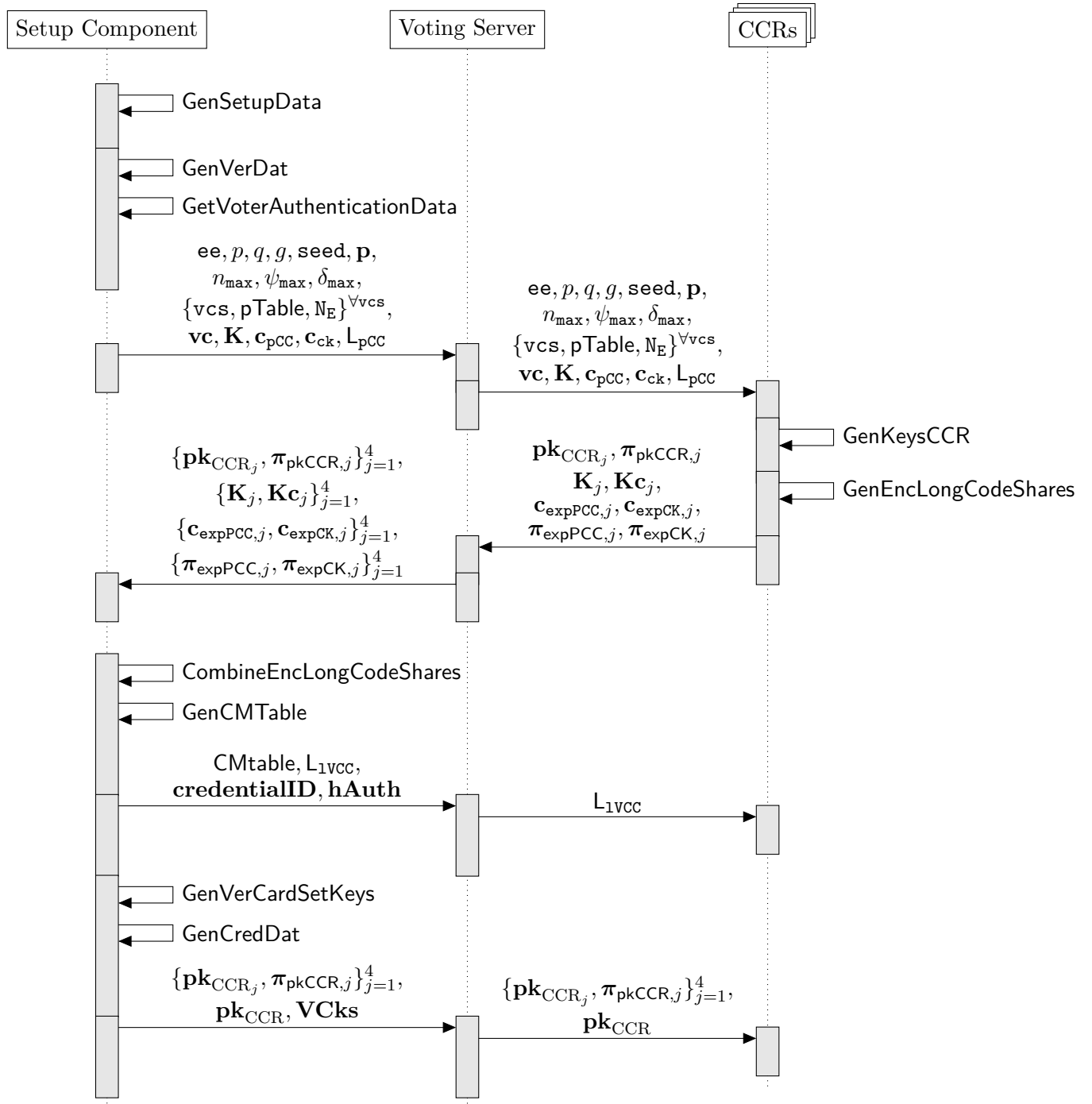


Fig. 7: Sequence diagram of the ~~SetupVoting~~SetupVoting protocol, after which the auditors run the ~~VerifyConfigPhase~~VerifyConfigPhase algorithm and each voter receives a Voting Card containing the codes.

4.1.1 GenSetupData

The setup component generates the election event context, the primes mapping table **pTable** and a key pair to encrypt the partial Choice Return Codes **pCC_{id}** during the configuration phase. **GenSetupData** ensures that the same actual voting option maps to the same prime number in all verification card sets. This is achieved by defining the initially empty key-value map **p_{map}** which is augmented during the **GenSetupData** algorithm with the key-value pairs (v_i, \tilde{p}_i) , where the actual voting option is the key and the prime is the corresponding value.

Algorithm 4.1 GenSetupData

Context:

Vector of verification card set IDs $\mathbf{vcs} = (\mathbf{vcs}_0, \dots, \mathbf{vcs}_{N_{bb}-1}) \in ((\mathbb{A}_{Base16})^{1_{ID}})^{N_{bb}}$

For each of the N_{bb} verification card sets $\mathbf{vcs} \in \mathbf{vcs}$ (see section 3.4)

Actual voting options $\tilde{\mathbf{v}} = (v_0, \dots, v_{n-1})$, $v_i \in \mathcal{T}_1^{50}$ \triangleright The actual voting options of this specific verification card set in the order defined by the election event configuration

Semantic information $\sigma = (\sigma_0, \dots, \sigma_{n-1})$, $\sigma_i \in \mathbb{A}_{UCS}^*$ \triangleright Matching order between actual voting options and semantic information

Correctness information $\tau = (\tau_0, \dots, \tau_{n-1})$, $\tau_i \in \mathcal{T}_1^{50}$ \triangleright Matching order between actual voting options and correctness information

Maximum supported number of voting options $n_{sup} \in \mathbb{N}^+$

\triangleright See section 3.5.1

Maximum supported number of selections $\psi_{sup} \in \mathbb{N}^+$

Maximum supported number of write-ins + 1: $\delta_{sup} \in \mathbb{N}^+$

Input:

Encryption parameter's seed: $\mathbf{seed} \in \mathbb{A}_{UCS}^{16}$

\triangleright The name of the election event in the format specified in section 3.2

Require: $1 \leq n \leq n_{sup}$, $\forall \mathbf{vcs} \in \mathbf{vcs}$

Operation:

```

1:  $(p, q, g, (p_0, \dots, p_{n_{sup}-1})) \leftarrow \text{GetElectionEventEncryptionParameters}(\mathbf{seed})$   $\triangleright$  See algorithm 3.1
2:  $\mathbf{pmap} \leftarrow ()$ 
3:  $k \leftarrow 0$ 
4:  $n_{max} \leftarrow 0$ 
5:  $\psi_{max} \leftarrow 0$ 
6:  $\delta_{max} \leftarrow 0$ 
7: for  $\mathbf{vcs} \in \mathbf{vcs}$  do  $\triangleright$  Iterating through  $\mathbf{vcs}$  in lexicographic order
8:   for  $i \in [0, n]$  do
9:     if  $v_i \in \mathbf{pmap}$  then  $\triangleright$  Look up if the value  $v_i$  exists as a key in  $\mathbf{pmap}$ 
10:        $\tilde{p}_i \leftarrow \mathbf{pmap}(v_i)$   $\triangleright$  Retrieve the value from the key-value map
11:     else
12:       if  $k \geq n_{sup}$  then
13:         return  $\perp$   $\triangleright$  The amount of distinct voting options across all verification card sets must not exceed  $n_{sup}$ 
14:       end if
15:        $\tilde{p}_i \leftarrow p_k$ 
16:        $\mathbf{pmap} \leftarrow \mathbf{pmap} \cup (v_i, \tilde{p}_i)$   $\mathbf{pmap} \leftarrow \mathbf{pmap} \cup (v_i, \tilde{p}_i)$ 
17:        $k \leftarrow k + 1$ 
18:     end if
19:   end for
20:    $\mathbf{pTable}_{\mathbf{vcs}} \leftarrow ((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ 
21:    $\psi \leftarrow \text{GetPsi}()$   $\triangleright$  See algorithm 3.8
22:    $\delta \leftarrow \text{GetDelta}()$   $\triangleright$  See algorithm 3.9
23:   if  $(\delta - 1 > \psi)$  then
24:     return  $\perp$   $\triangleright$  The number of write-ins of a verification card set must not exceed the number of selections
25:   end if
26:    $n_{max} \leftarrow \max(n_{max}, n)$ 
27:    $\psi_{max} \leftarrow \max(\psi_{max}, \psi)$ 
28:    $\delta_{max} \leftarrow \max(\delta_{max}, \delta)$ 
29: end for
30: if  $(\psi_{max} > \psi_{sup}) \vee (\delta_{max} > \delta_{sup})$  then
31:   return  $\perp$   $\triangleright$  The maximum number of selections or write-ins must not exceed the supported values
32: end if
33:  $\mathbf{pTable} \leftarrow (\mathbf{pTable}_{\mathbf{vcs}_0}, \dots, \mathbf{pTable}_{\mathbf{vcs}_{N_{bb}-1}})$   $\triangleright$  Each verification card set has its  $\mathbf{pTable}$ 
34:  $(\mathbf{pk}_{setup}, \mathbf{sk}_{setup}) \leftarrow \text{GenKeyPair}(p, q, g, n_{max})$   $\triangleright$  See crypto primitives specification

```

Output:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Small primes $\mathbf{p} = (p_0, \dots, p_{n_{sup}-1})$, $p_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g, (p_i < p_{i+1}) \forall i \in [0, n_{sup})$

Maximum number of voting options $n_{max} \in [1, n_{sup}]$

Maximum number of selections $\psi_{max} \in [1, \psi_{sup}]$

Maximum number of write-ins + 1: $\delta_{max} \in [1, \delta_{sup}]$

Prime mapping tables for each verification card set \mathbf{pTable}

Setup public key $\mathbf{pk}_{setup} \in \mathbb{G}_q^{n_{max}}$

Setup secret key $\mathbf{sk}_{setup} \in \mathbb{Z}_q^{n_{max}}$

4.1.2 GenVerDat

The setup component runs the GenVerDat (Generate Verification Data) algorithm to initialize the control components' computation of the return codes.

Algorithm 4.2 GenVerDat

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Number of eligible voters for this verification card set $N_E \in \mathbb{N}^+$
 Primes mapping table $pTable \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ \triangleright $pTable$ is of the form
 $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$
 Maximum number of voting options $n_{max} \in [1, n_{sup}]$

Input:

Setup public key $pk_{setup} \in \mathbb{G}_q^{n_{max}}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1:  $L_{pcc} \leftarrow \{\}$   $L_{pcc} \leftarrow ()$ 
2:  $\tilde{p} = (\tilde{p}_0, \dots, \tilde{p}_{n-1}) \leftarrow \text{GetEncodedVotingOptions}()$   $\triangleright$  See Algorithm 3.2
3:  $\tau = (\tau_0, \dots, \tau_{n-1}) \leftarrow \text{GetCorrectnessInformation}()$   $\triangleright$  See Algorithm 3.5
4: for  $id \in [0, N_E]$  do
5:    $vc_{id} \leftarrow \text{GenRandomString}(1_{ID}, \mathbb{A}_{Base16})$ 
6:    $SVK_{id} \leftarrow \text{GenRandomString}(1_{SVK}, \mathbb{A}_{u32})$ 
7:    $(K_{id}, k_{id}) \leftarrow \text{GenKeyPair}(p, q, g, 1)$ 
8:   for  $k \in [0, n]$  do
9:      $pCC_{id,k} \leftarrow \tilde{p}_k^{k_{id}} \bmod p$ 
10:     $hpCC_{id,k} \leftarrow \text{HashAndSquare}(pCC_{id,k})$ 
11:     $lpCC_{id,k} \leftarrow \text{RecursiveHash}(hpCC_{id,k}, vc_{id}, ee, \tau_k)$ 
12:     $L_{pcc} \leftarrow L_{pcc} \cup \{\text{Base64Encode}(lpCC_{id,k})\}$   $L_{pcc} \leftarrow L_{pcc} \parallel \text{Base64Encode}(lpCC_{id,k})$ 
13:  end for
14:   $hpCC_{id} \leftarrow (hpCC_{id,0}, \dots, hpCC_{id,n-1})$ 
15:   $c_{pcc,id} \leftarrow \text{GetCiphertext}(hpCC_{id}, \text{GenRandomInteger}(q), pk_{setup})$ 
16:  do
17:     $BCK_{id} \leftarrow \text{GenUniqueDecimalStrings}(1_{BCK}, 1)$   $\triangleright$  Assign first element of list to  $BCK_{id}$ 
18:    while  $\text{StringToInteger}(BCK_{id}) = 0$ 
19:       $hBCK_{id} \leftarrow \text{HashAndSquare}(\text{StringToInteger}(BCK_{id}))$ 
20:       $CK_{id} \leftarrow hBCK_{id}^{k_{id}} \bmod p$ 
21:       $hCK_{id} \leftarrow \text{HashAndSquare}(CK_{id})$ 
22:       $c_{ck,id} \leftarrow \text{GetCiphertext}(hCK_{id}, \text{GenRandomInteger}(q), pk_{setup})$ 
23:  end for
24:  $L_{pcc} \leftarrow \text{Order}(L_{pcc})$   $\triangleright$  Lexicographic ordering of the list ensures the unlinkability to the order of insertion.
```

Output:

Vector of verification card IDs $vc = (vc_0, \dots, vc_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
 Vector of Start Voting Keys $SVK = (SVK_0, \dots, SVK_{N_E-1}) \in ((\mathbb{A}_{u32})^{1_{SVK}})^{N_E}$
 Vector of verification card public keys $K = (K_0, \dots, K_{N_E-1}) \in \mathbb{G}_q^{N_E}$
 Vector of verification card secret keys $k = (k_0, \dots, k_{N_E-1}) \in \mathbb{Z}_q^{N_E}$
 Partial Choice Return Codes allow list $L_{pcc} \in (\mathbb{A}_{Base64})^{1_{HB64} \times n \cdot N_E}$ \triangleright Order of elements uncorrelated to order of insertion
 Vector of ballot casting keys $BCK = (BCK_0, \dots, BCK_{N_E-1}) \in ((\mathbb{A}_{10})^{1_{BCK}})^{N_E}$
 Vector of encrypted, hashed partial Choice Return Codes $c_{pcc} = (c_{pcc,0}, \dots, c_{pcc,N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
 Vector of encrypted, hashed Confirmation Keys $c_{ck} = (c_{ck,0}, \dots, c_{ck,N_E-1}) \in (\mathbb{G}_q^2)^{N_E}$

4.1.3 GetVoterAuthenticationData

The setup component runs the **GetVoterAuthenticationData** algorithm to generate the data for the voter authentication protocol between the voting client and the voting server.

The cantons provide an extended authentication factor—usually the voter’s birth date or year—to the Swiss Post Voting System. Although this factor only offers limited security value due to its low entropy and potential exposure through sources like social media, it nonetheless enhances authentication:

- The extended authentication factor serves as a minor psychological hurdle for individuals attempting voter impersonation.
- The voter and cantons are the primary holders of this information. As such, the operators of the voting server, control or printing components must exert considerable effort to learn the date or year of birth from a large voter base.
- Its inclusion maintains consistency with previous online voting systems in Switzerland, which also utilized this extended authentication factor.

Hence, despite its limitations, using an extended authentication factor has some advantages compared to not having one at all.

Algorithm 4.3 GetVoterAuthenticationData

Context:

Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Number of eligible voters for this verification card set $N_E \in \mathbb{N}^+$

Character length of the extended authentication factor $1_{EA} \in \mathbb{N}^+$

Input:

Vector of Start Voting Keys $\mathbf{SVK} = (\mathbf{SVK}_0, \dots, \mathbf{SVK}_{N_E-1}) \in ((\mathbb{A}_{u32})^{1_{SVK}})^{N_E}$

Vector of extended authentication factors $\mathbf{EA} = (\mathbf{EA}_0, \dots, \mathbf{EA}_{N_E-1}) \in (\mathbb{A}_{10}^{1_{EA}})^{N_E}$

Operation:

- 1: **for** $\mathbf{id} \in [0, N_E)$ **do**
 - 2: $\mathbf{credentialID}_{\mathbf{id}} \leftarrow \text{DeriveCredentialId}(\mathbf{SVK}_{\mathbf{id}})$ ▷ See algorithm 3.19
 - 3: $\mathbf{hAuth}_{\mathbf{id}} \leftarrow \text{DeriveBaseAuthenticationChallenge}(\mathbf{SVK}_{\mathbf{id}}, \mathbf{EA}_{\mathbf{id}})$ ▷ See algorithm 3.20
 - 4: **end for**
 - 5: $\mathbf{credentialID} \leftarrow (\mathbf{credentialID}_0 \dots, \mathbf{credentialID}_{N_E-1})$
 - 6: $\mathbf{hAuth} \leftarrow (\mathbf{hAuth}_0 \dots, \mathbf{hAuth}_{N_E-1})$
-

Output:

Vector of derived voter identifiers $\mathbf{credentialID} \in ((\mathbb{A}_{Base16})^{1_{ID}})^{N_E}$

Vector of base authentication challenges $\mathbf{hAuth} \in ((\mathbb{A}_{Base64})^{1_{HB64}})^{N_E}$

After running the algorithms 4.1, 4.2 and 4.3, the setup component sends the following information to the Return Codes control components CCR.

- The list of verification card IDs \mathbf{vc}
- The vector of verification card public keys \mathbf{K}
- The vector of encrypted, hashed, squared partial Choice Return Codes \mathbf{c}_{pcc}
- The list of encrypted, hashed, squared confirmation keys \mathbf{c}_{ck}
- The partial Choice Return Codes allow list \mathbf{L}_{pcc}

The amount of data increases with the number of voting options n and the number of eligible voters N_E of the verification card set; therefore, the implementation should divide the data into chunks to keep the size manageable.

4.1.4 GenKeysCCR

The Return Codes control components CCR generate the CCR_j Choice Return Codes encryption key pair $\mathbf{pk}_{\text{CCR}_j}, \mathbf{sk}_{\text{CCR}_j}$ to encrypt the partial Choice Return Codes \mathbf{pCC}_{id} during the voting phase and the CCR_j Return Codes Generation secret key \mathbf{k}'_j .

Algorithm 4.4 GenKeysCCR

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 The CCR's index $j \in [1, 4]$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{\text{Base16}})^{1\text{ID}}$
 Maximum number of selections $\psi_{\text{max}} \in [1, \psi_{\text{sup}}]$

Operation:

▷ For all algorithms see the crypto primitives specification

- 1: $(\mathbf{pk}_{\text{CCR}_j}, \mathbf{sk}_{\text{CCR}_j}) \leftarrow \text{GenKeyPair}(p, q, g, \psi_{\text{max}})$
 - 2: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{ee}, \text{"GenKeysCCR"}, \text{IntegerToString}(j))$
 - 3: **for** $i \in [0, \psi_{\text{max}})$ **do**
 - 4: $\pi_{\mathbf{pk}_{\text{CCR}_j}, i} \leftarrow \text{GenSchnorrProof}(\mathbf{sk}_{\text{CCR}_j}, \mathbf{pk}_{\text{CCR}_j}, \mathbf{i}_{\text{aux}})$
 - 5: **end for**
 - 6: $\boldsymbol{\pi}_{\mathbf{pk}_{\text{CCR}_j}} \leftarrow (\pi_{\mathbf{pk}_{\text{CCR}_j}, 0}, \dots, \pi_{\mathbf{pk}_{\text{CCR}_j}, \psi_{\text{max}}-1})$
 - 7: $\mathbf{k}'_j \leftarrow \text{GenRandomInteger}(q)$
-

Output:

CCR_j Choice Return Codes encryption public key $\mathbf{pk}_{\text{CCR}_j} = (\mathbf{pk}_{\text{CCR}_j, 0}, \dots, \mathbf{pk}_{\text{CCR}_j, \psi_{\text{max}}-1}) \in \mathbb{G}_q^{\psi_{\text{max}}}$
 CCR_j Choice Return Codes encryption secret key $\mathbf{sk}_{\text{CCR}_j} = (\mathbf{sk}_{\text{CCR}_j, 0}, \dots, \mathbf{sk}_{\text{CCR}_j, \psi_{\text{max}}-1}) \in \mathbb{Z}_q^{\psi_{\text{max}}}$
 CCR_j Return Codes Generation secret key $\mathbf{k}'_j \in \mathbb{Z}_q$
 CCR_j Schnorr proofs of knowledge $\boldsymbol{\pi}_{\mathbf{pk}_{\text{CCR}_j}} = (\pi_{\mathbf{pk}_{\text{CCR}_j}, 0}, \dots, \pi_{\mathbf{pk}_{\text{CCR}_j}, \psi_{\text{max}}-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi_{\text{max}}}$

4.1.5 GenEncLongCodeShares

Each Return Codes control component CCR_j runs the **GenEncLongCodeShares** (Generate Encrypted Long Return Code Shares) to create shares of the long return codes. The main idea behind this algorithm is to derive a secret key per voter and to exponentiate the setup component's input. **GenEncLongCodeShares** is analogous to the voting phase's algorithms **CreateLCCShare** (See algorithm 5.8) and **CreateLVCCShare** (algorithm 5.11).

The Return Codes control component CCR_j derives two different keys from the CCR_j Return Codes Generation secret key k'_j :

- Voter Choice Return Code Generation key pair $K_{j,id}, k_{j,id}$
- Voter Vote Cast Return Code Generation key pair $Kc_{j,id}, kc_{j,id}$

Using different keys for generating the control component's shares of the long Choice Return Codes and the long Vote Cast Return Codes prevents an attacker from abusing some confirmation attempts as a way to glean Choice Return Codes.

The control components store the list of processed voting cards $L_{genVC,j}$, which is initialized to the empty list before the first execution of the algorithm.

Moreover, the control components initialize the following lists for all processed voting cards:

- list of voting cards $L_{decPCC,j}$ for which the control components decrypted the partial Choice Return Codes pCC_{id} ,
- list of voting cards $L_{sentVotes,j}$ for which the control components already generated long Choice Return Code shares,
- list of confirmed votes $L_{confirmedVotes,j}$,
- key-value map of number of confirmation attempts per voting card $L_{confirmationAttempts,j}$; initially with all 0-values.

At the end of the **SetupTallyEB** algorithm of the **SetupTally** protocol, the setup component sends the public keys to the control components.

The control components verify that the election event context is consistent with the information processed in **GenEncLongCodeShares**. In particular, the control components check the following:

- The number of verification cards for each verification card set corresponds to the number of verification cards processed in **GenEncLongCodeShares**.
- The correctness of the setup and other control components' key generation by executing the **VerifyKeyGenerationSchnorrProofs** algorithm.

Algorithm 4.5 GenEncLongCodeShares

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 The CCR's index $j \in [1, 4]$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Vector of verification card IDs $\mathbf{vc} = (\mathbf{vc}_0, \dots, \mathbf{vc}_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
 Number of voting options for this verification card set $n \in [1, n_{sup}] \triangleright$ Can be derived from \mathbf{pTable}

Stateful Lists and Maps:

List of generated voting cards $\mathbf{L}_{genVC,j}$

Input:

CCR _{j} Return Codes Generation secret key $\mathbf{k}'_j \in \mathbb{Z}_q$
 Vector of encrypted, hashed partial Choice Return Codes $\mathbf{c}_{PCC} = (\mathbf{c}_{PCC,0}, \dots, \mathbf{c}_{PCC,N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
 Vector of encrypted, hashed Confirmation Keys $\mathbf{c}_{CK} = (\mathbf{c}_{CK,0}, \dots, \mathbf{c}_{CK,N_E-1}) \in (\mathbb{G}_q^2)^{N_E}$

Require: $\mathbf{vc}_{id} \notin \mathbf{L}_{genVC,j} \ \forall \ \mathbf{vc}_{id} \in \mathbf{vc}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: PRK  $\leftarrow$  IntegerToByteArray( $\mathbf{k}'_j$ )
2: for  $\mathbf{id} \in [0, N_E)$  do
3:    $\mathbf{info} \leftarrow$  ("VoterChoiceReturnCodeGeneration",  $\mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{id}$ )
4:    $\mathbf{k}_{j,id} \leftarrow$  KDFToZq(PRK,  $\mathbf{info}, q$ )
5:    $\mathbf{K}_{j,id} \leftarrow g^{\mathbf{k}_{j,id}} \bmod p$ 
6:    $\mathbf{info}_{CK} \leftarrow$  ("VoterVoteCastReturnCodeGeneration",  $\mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{id}$ )
7:    $\mathbf{kc}_{j,id} \leftarrow$  KDFToZq(PRK,  $\mathbf{info}_{CK}, q$ )
8:    $\mathbf{Kc}_{j,id} \leftarrow g^{\mathbf{kc}_{j,id}} \bmod p$ 
9:    $\mathbf{c}_{expPCC,j,id} = (\mathbf{c}_{expPCC,j,id,0}, \dots, \mathbf{c}_{expPCC,j,id,n}) \leftarrow$  GetCiphertextExponentiation( $\mathbf{c}_{PCC,id}, \mathbf{k}_{j,id}$ )
10:   $\mathbf{i}_{aux} \leftarrow$  ( $\mathbf{ee}, \mathbf{vc}_{id}$ , "GenEncLongCodeShares", IntegerToString( $j$ ))
11:   $\pi_{expPCC,j,id} \leftarrow$  GenExponentiationProof( $((g, \mathbf{c}_{PCC,id}), \mathbf{k}_{j,id}, (\mathbf{K}_{j,id}, \mathbf{c}_{expPCC,j,id}), \mathbf{i}_{aux})$ )
12:   $\mathbf{c}_{expCK,j,id} = (\mathbf{c}_{expCK,j,id,0}, \dots, \mathbf{c}_{expCK,j,id,n}) \leftarrow$  GetCiphertextExponentiation( $\mathbf{c}_{CK,id}, \mathbf{kc}_{j,id}$ )
13:   $\pi_{expCK,j,id} \leftarrow$  GenExponentiationProof( $((g, \mathbf{c}_{CK,id}), \mathbf{kc}_{j,id}, (\mathbf{Kc}_{j,id}, \mathbf{c}_{expCK,j,id}), \mathbf{i}_{aux})$ )
14:   $\mathbf{L}_{genVC,j} \leftarrow \mathbf{L}_{genVC,j} \cup \mathbf{vc}_{id}$   $\mathbf{L}_{genVC,j} \leftarrow \mathbf{L}_{genVC,j} \parallel \mathbf{vc}_{id}$ 
15: end for

```

Output:

Vector of Voter Choice Return Code Generation public keys $\mathbf{K}_j = (\mathbf{K}_{j,0}, \dots, \mathbf{K}_{j,N_E-1}) \in \mathbb{G}_q^{N_E}$
 Vector of Voter Vote Cast Return Code Generation public keys $\mathbf{Kc}_j = (\mathbf{Kc}_{j,0}, \dots, \mathbf{Kc}_{j,N_E-1}) \in \mathbb{G}_q^{N_E}$
 Vector of exponentiated, encrypted, hashed partial Choice Return Codes $\mathbf{c}_{expPCC,j} = (\mathbf{c}_{expPCC,j,0}, \dots, \mathbf{c}_{expPCC,j,N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
 Proofs of correct exponentiation of the partial Choice Return Codes $\pi_{expPCC,j} = (\pi_{expPCC,j,0}, \dots, \pi_{expPCC,j,N_E-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E}$
 Vector of exponentiated, encrypted, hashed Confirmation Keys $\mathbf{c}_{expCK,j} = (\mathbf{c}_{expCK,j,0}, \dots, \mathbf{c}_{expCK,j,N_E-1}) \in (\mathbb{G}_q^2)^{N_E}$
 Proofs of correct exponentiation of the Confirmation Keys $\pi_{expCK,j} = (\pi_{expCK,j,0}, \dots, \pi_{expCK,j,N_E-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E}$

Subsequently, the control components send their encrypted return code shares back to the setup component. Upon reception, the setup component verifies the received zero-knowledge

proofs and checks the consistency of the received information, notably that the number of encrypted return code shares corresponds to the correct number of verification cards.

~~For performance reasons, the setup component delegates the verification of the exponentiation proofs to the verifier in the VerifyConfigPhase. The verifier runs in the same controlled offline environment in the configuration phase. Hence, the delegation of the verification of the zero-knowledge proof allows parallelization of time-consuming operations for generating the return codes.—~~

4.1.6 CombineEncLongCodeShares

The setup component combines the control components' encrypted long return code shares, after having verified the zero-knowledge proofs. If at least one of the verifications fails, the algorithm fails. Moreover, the setup component generates the long Vote Cast Return Codes allow list L_{1vcc} that is used during the voting phase in the algorithm 5.12.

Algorithm 4.6 CombineEnLongCodeShares

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Vector of verification card IDs $\mathbf{vc} = (\mathbf{vc}_0, \dots, \mathbf{vc}_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
 Number of voting options for this verification card set $n \in [1, n_{\max}]$ ▷ Can be derived from pTable
 Maximum number of voting options $n_{\max} \in [1, n_{\sup}]$

Input:

Setup secret key $\mathbf{sk}_{\text{setup}} \in \mathbb{Z}_q^{n_{\max}}$
Vector of encrypted, hashed partial Choice Return Codes $\mathbf{c}_{\text{PCC}} = (\mathbf{c}_{\text{PCC},0}, \dots, \mathbf{c}_{\text{PCC},N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
Vector of encrypted, hashed Confirmation Keys $\mathbf{c}_{\text{CK}} = (\mathbf{c}_{\text{CK},0}, \dots, \mathbf{c}_{\text{CK},N_E-1}) \in (\mathbb{G}_q^2)^{N_E}$
Control components' vectors of Voter Choice Return Code Generation public keys $(\mathbf{K}_1, \mathbf{K}_2, \mathbf{K}_3, \mathbf{K}_4) \in (\mathbb{G}_q^{N_E})^4$
Control components' vectors of Voter Vote Cast Return Code Generation public keys $(\mathbf{Kc}_1, \mathbf{Kc}_2, \mathbf{Kc}_3, \mathbf{Kc}_4) \in (\mathbb{G}_q^{N_E})^4$
 Matrix of exponentiated, encrypted, hashed partial Choice Return Codes $\mathbf{C}_{\text{expPCC}} = (\mathbf{c}_{\text{expPCC},1}, \dots, \mathbf{c}_{\text{expPCC},4}) \in ((\mathbb{G}_q^{n+1})^{N_E})^4$
Control components' proofs of correct exponentiation of the partial Choice Return Codes
 $(\pi_{\text{expPCC},1}, \pi_{\text{expPCC},2}, \pi_{\text{expPCC},3}, \pi_{\text{expPCC},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E})^4$
 Matrix of exponentiated, encrypted, hashed Confirmation Keys $\mathbf{C}_{\text{expCK}} = (\mathbf{c}_{\text{expCK},1}, \dots, \mathbf{c}_{\text{expCK},4}) \in ((\mathbb{G}_q^2)^{N_E})^4$
Control components' proofs of correct exponentiation of the Confirmation Keys
 $(\pi_{\text{expCK},1}, \pi_{\text{expCK},2}, \pi_{\text{expCK},3}, \pi_{\text{expCK},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E})^4$

Operation:

▷ For all algorithms see the crypto primitives specification

```

1: for  $j \in [1, 4]$  do
2:    $\mathbf{lvcc} \leftarrow \{\}$   $\pi_{\text{ExpPccVerif}} \leftarrow \text{VerifyEncryptedPCCExponentiationProofs}(\mathbf{c}_{\text{PCC}}, \mathbf{K}_j, \mathbf{c}_{\text{expPCC},j}, \pi_{\text{expPCC},j})$  ▷ See algorithm 4.7
3:    $\pi_{\text{ExpCkVerif}} \leftarrow \text{VerifyEncryptedCKExponentiationProofs}(\mathbf{c}_{\text{CK}}, \mathbf{Kc}_j, \mathbf{c}_{\text{expCK},j}, \pi_{\text{expCK},j})$  ▷ See algorithm 4.8
4:   if  $\neg \pi_{\text{ExpPccVerif}} \vee \neg \pi_{\text{ExpCkVerif}}$  then
5:     return  $\perp$ 
6:   end if
7: end for
8:  $\mathbf{lvcc} \leftarrow \{\}$ 
9: for  $\text{id} \in [0, N_E]$  do
10:   $\mathbf{c}_{\text{P},\text{id}} \leftarrow (1, \{1\}^n)$  ▷ Neutral element of ciphertext multiplication
11:  for  $j \in [1, 4]$  do
12:     $\mathbf{c}_{\text{P},\text{id}} \leftarrow \text{GetCiphertextProduct}(\mathbf{c}_{\text{P},\text{id}}, \mathbf{c}_{\text{expPCC},j,\text{id}})$ 
13:     $\mathbf{lvcc}_{j,\text{id}} \leftarrow \text{GetMessage}(\mathbf{c}_{\text{expCK},j,\text{id}}, \mathbf{sk}_{\text{setup}})$ 
14:     $\mathbf{i}_{\text{aux},1} \leftarrow (\text{"CreateLVCCShare"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{\text{id}}, \text{IntegerToString}(j))$ 
15:     $\mathbf{hlvcc}_{j,\text{id}} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(\mathbf{i}_{\text{aux},1}, \mathbf{lvcc}_{j,\text{id}}))$ 
16:  end for
17:   $\mathbf{pVCC}_{\text{id}} \leftarrow \prod_{j=1}^4 \mathbf{lvcc}_{j,\text{id}} \bmod p$ 
18:   $\mathbf{i}_{\text{aux},2} \leftarrow (\text{"VerifyLVCCHash"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{\text{id}})$ 
19:   $\mathbf{hhlVCC}_{\text{id}} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(\mathbf{i}_{\text{aux},2}, \mathbf{hlvcc}_{1,\text{id}}, \mathbf{hlvcc}_{2,\text{id}}, \mathbf{hlvcc}_{3,\text{id}}, \mathbf{hlvcc}_{4,\text{id}}))$ 
20:   $\mathbf{lvcc} \leftarrow \mathbf{lvcc} \cup (\mathbf{hhlVCC}_{\text{id}}) \mathbf{lvcc} \leftarrow \mathbf{lvcc} \parallel \mathbf{hhlVCC}_{\text{id}}$ 
21: end for

```

Output:

22: if *all* control components' zero-knowledge proofs verify then
 Vector of encrypted pre-Choice Return Codes $\mathbf{c}_{\text{PC}} = (\mathbf{c}_{\text{PC},0}, \dots, \mathbf{c}_{\text{PC},N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
 Pre-Vote Cast Return Codes $\mathbf{pVCC} = (\mathbf{pVCC}_0, \dots, \mathbf{pVCC}_{N_E-1}) \in (\mathbb{G}_q)^{N_E}$
 Long Vote Cast Return Codes allow list $\mathbf{L}_{\text{lvcc}} \in (\mathbb{A}_{Base64}^{1_{HB64}})^{N_E}$
 23: else
 \perp , the verification of at least one $\pi_{\text{expPCC},j,\text{id}}$ or $\pi_{\text{expCK},j,\text{id}}$ failed.
 24: end if

The verifications in Algorithm 4.6 rely on the following algorithms.

Algorithm 4.7 VerifyEncryptedPCCExponentiationProofs

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
The CCR's index $j \in [1, 4]$
Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
Vector of verification card IDs $\mathbf{vc} = (vc_0, \dots, vc_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
Number of voting options for this verification card set $n \in [1, n_{sup}]$

Input:

Encrypted, hashed partial Choice Return Codes $\mathbf{c}_{pcc} \in (\mathbb{G}_q^{n+1})^{N_E}$
Voter Choice Return Code Generation public keys $\mathbf{K}_j \in \mathbb{G}_q^{N_E}$
Exponentiated, encrypted, hashed partial Choice Return Codes $\mathbf{c}_{expPCC,j} \in (\mathbb{G}_q^{n+1})^{N_E}$
Proofs of correct exponentiation $\pi_{expPCC,j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E}$

Operation:

```

1: for id  $\in [0, N_E)$  do
2:    $\mathbf{g} \leftarrow (g, \mathbf{c}_{pcc,id})$  ▷ The bases
3:    $\mathbf{y} \leftarrow (K_{j,id}, \mathbf{c}_{expPCC,j,id})$  ▷ The exponentiations
4:    $\mathbf{i}_{aux} \leftarrow (ee, \mathbf{vc}_{id}, \text{"GenEncLongCodeShares"}, \text{IntegerToString}(j))$  ▷ See crypto primitives specification
5:    $\text{exponentiationVerif}_{id} \leftarrow \text{VerifyExponentiation}(\mathbf{g}, \mathbf{y}, \pi_{expPCC,j,id}, \mathbf{i}_{aux})$  ▷ See crypto primitives specification
6: end for
7: if  $\text{exponentiationVerif}_{id} \forall id$  then
8:   return  $\top$ 
9: else
10:  return  $\perp$ 
11: end if

```

Output:

The result of the verification: \top if the verification is successful for *this specific* verification card set, \perp otherwise.

Algorithm 4.8 VerifyEncryptedCKExponentiationProofs

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
The CCR's index $j \in [1, 4]$
Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
Vector of verification card IDs $\mathbf{vc} = (vc_0, \dots, vc_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$

Input:

Encrypted, hashed Confirmation Key $\mathbf{c}_{ck} \in (\mathbb{G}_q^2)^{N_E}$
Voter Vote Cast Return Code Generation public keys $\mathbf{Kc}_j \in \mathbb{G}_q^{N_E}$
Exponentiated, encrypted, hashed Confirmation Key $\mathbf{c}_{expCK,j} \in (\mathbb{G}_q^2)^{N_E}$
Proofs of correct exponentiation $\pi_{expCK,j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{N_E}$

Operation:

```

1: for  $id \in [0, N_E)$  do
2:    $\mathbf{g} \leftarrow (g, \mathbf{c}_{ck, id})$  ▷ The bases
3:    $\mathbf{y} \leftarrow (\mathbf{Kc}_{j, id}, \mathbf{c}_{expCK, j, id})$  ▷ The exponentiations
4:    $\mathbf{i}_{aux} \leftarrow (ee, \mathbf{vc}_{id}, \text{"GenEncLongCodeShares"}, \text{IntegerToString}(j))$  ▷ See crypto primitives
   specification
5:    $\text{exponentiationVerif}_{id} \leftarrow \text{VerifyExponentiation}(\mathbf{g}, \mathbf{y}, \pi_{expCK, j, id}, \mathbf{i}_{aux})$  ▷ See crypto
   primitives specification
6: end for
7: if  $\text{exponentiationVerif}_{id} \forall id$  then
8:   return  $\top$ 
9: else
10:  return  $\perp$ 
11: end if

```

Output:

The result of the verification: \top if the verification is successful for *this specific* verification card set, \perp otherwise.

4.1.7 GenCMTable

Subsequently, the setup component generates the Return Codes Mapping table **CMtable** that allows the control components and the voting server to retrieve the short Choice Return Codes **CC_{id}**. At the end of the algorithm, we order the Return Codes Mapping table **CMtable** by the hash of the long Return Codes to break the correlation between the voting option and the order of insertion.

Algorithm 4.9 GenCMTable

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Vector of verification card IDs $\mathbf{vc} = (\mathbf{vc}_0, \dots, \mathbf{vc}_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
 Correctness information $\tau = (\tau_0, \dots, \tau_{n-1}) \in (\mathcal{T}_1^{50})^n$ \triangleright Can be derived from **pTable** using algorithm 3.5
 Maximum number of voting options $n_{\max} \in [1, n_{\sup}]$

Input:

Setup secret key $\mathbf{sk}_{\text{setup}} \in \mathbb{Z}_q^{n_{\max}}$
 Vector of encrypted pre-Choice Return Codes $\mathbf{c}_{pC} = (c_{pC,0}, \dots, c_{pC,N_E-1}) \in (\mathbb{G}_q^{n+1})^{N_E}$
 Pre-Vote Cast Return Codes $\mathbf{pVCC} = (\mathbf{pVCC}_0, \dots, \mathbf{pVCC}_{N_E-1}) \in (\mathbb{G}_q)^{N_E}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: CMtable  $\leftarrow \{\}$  CMtable  $\leftarrow ()$ 
2: for  $id \in [0, N_E)$  do
3:    $\mathbf{CC}_{id} \leftarrow \text{GenUniqueDecimalStrings}(1_{CC}, n)$ 
4:    $\mathbf{pC}_{id} \leftarrow \text{GetMessage}(c_{pC,id}, \mathbf{sk}_{\text{setup}})$ 
5:   for  $k \in [0, n)$  do
6:      $1_{CC_{id,k}} \leftarrow \text{RecursiveHash}(\mathbf{pC}_{id,k}, \mathbf{vc}_{id}, ee, \tau_k)$ 
7:      $\mathbf{sk}_{cc_{id,k}} \leftarrow \text{KDF}(1_{CC_{id,k}}, (), 1_{KD})$ 
8:      $(\mathbf{ct}_{CC_{id,k}, \text{ciphertext}}, \mathbf{ct}_{CC_{id,k}, \text{nonce}}) \leftarrow \text{GenCiphertextSymmetric}(\mathbf{sk}_{cc_{id,k}}, \text{StringToByteArray}(\mathbf{CC}_{id,k}), ())$ 
9:     CMtable  $\leftarrow \mathbf{CMtable} \cup \{\text{Base64Encode}(\text{RecursiveHash}(1_{CC_{id,k}})), \text{Base64Encode}(\mathbf{ct}_{CC_{id,k}, \text{ciphertext}} || \mathbf{ct}_{CC_{id,k}, \text{nonce}})\}$ 
    CMtable  $\leftarrow \mathbf{CMtable} || (\text{Base64Encode}(\text{RecursiveHash}(1_{CC_{id,k}})), \text{Base64Encode}(\mathbf{ct}_{CC_{id,k}, \text{ciphertext}} || \mathbf{ct}_{CC_{id,k}, \text{nonce}}))$ 
10:  end for
11:   $1_{VCC_{id}} \leftarrow \text{RecursiveHash}(\mathbf{pVCC}_{id}, \mathbf{vc}_{id}, ee)$ 
12:   $\mathbf{VCC}_{id} \leftarrow \text{GenUniqueDecimalStrings}(1_{VCC}, 1)$   $\triangleright$  Assign first element of list to  $\mathbf{VCC}_{id}$ 
13:   $\mathbf{sk}_{vcc_{id}} \leftarrow \text{KDF}(1_{VCC_{id}}, (), 1_{KD})$ 
14:   $(\mathbf{ct}_{VCC_{id}, \text{ciphertext}}, \mathbf{ct}_{VCC_{id}, \text{nonce}}) \leftarrow \text{GenCiphertextSymmetric}(\mathbf{sk}_{vcc_{id}}, \text{StringToByteArray}(\mathbf{VCC}_{id}), ())$ 
15:  CMtable  $\leftarrow \mathbf{CMtable} \cup \{\text{Base64Encode}(\text{RecursiveHash}(1_{VCC_{id}})), \text{Base64Encode}(\mathbf{ct}_{VCC_{id}, \text{ciphertext}} || \mathbf{ct}_{VCC_{id}, \text{nonce}})\}$ 
    CMtable  $\leftarrow \mathbf{CMtable} || (\text{Base64Encode}(\text{RecursiveHash}(1_{VCC_{id}})), \text{Base64Encode}(\mathbf{ct}_{VCC_{id}, \text{ciphertext}} || \mathbf{ct}_{VCC_{id}, \text{nonce}}))$ 
16: end for
17: Order(CMtable, 1)  $\triangleright$  Lexicographic ordering by the table's first column (containing the encoded hash of the long Return Codes) ensures the unlinkability to the order of insertion.
```

Output:

Return Codes Mapping table $\mathbf{CMtable} \in (\mathbb{A}_{Base64}^{1_{HB64}} \times \mathbb{A}_{Base64}^{*})^{N_E \cdot (n+1)}$
 Vector of short Choice Return Codes $(\mathbf{CC}_0, \dots, \mathbf{CC}_{N_E-1}) \in ((\mathbb{A}_{10}^{1_{CC}})^n)^{N_E}$
 Vector of short Vote Cast Return Codes $(\mathbf{VCC}_0, \dots, \mathbf{VCC}_{N_E-1}) \in (\mathbb{A}_{10}^{1_{VCC}})^{N_E}$

The generated **CMtable** is a key-value map with keys $\mathbf{key} \in \mathbb{A}_{Base64}^{1_{HB64}}$ and corresponding values $\mathbf{value} \in \mathbb{A}_{Base64}^{*}$.

4.1.8 GenVerCardSetKeys

The setup component generates the verification card set keys by combining the CCR Choice Return Codes encryption public keys.

Algorithm 4.10 GenVerCardSetKeys

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Maximum number of selections $\psi_{\max} \in [1, \psi_{\sup}]$

Input:

CCR Choice Return Codes encryption public keys $\mathbf{pk}_{CCR} = ((\mathbf{pk}_{CCR_{1,0}}, \dots, \mathbf{pk}_{CCR_{1,\psi_{\max}-1}}), \dots, (\mathbf{pk}_{CCR_{4,0}}, \dots, \mathbf{pk}_{CCR_{4,\psi_{\max}-1}})) \in (\mathbb{G}_q^{\psi_{\max}})^4$
 CCR Schnorr proofs of knowledge $\boldsymbol{\pi}_{\mathbf{pk}_{CCR}} = (\boldsymbol{\pi}_{\mathbf{pk}_{CCR},1}, \boldsymbol{\pi}_{\mathbf{pk}_{CCR},2}, \boldsymbol{\pi}_{\mathbf{pk}_{CCR},3}, \boldsymbol{\pi}_{\mathbf{pk}_{CCR},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi_{\max}})^4$

Operation:

- 1: $\text{VerifSch} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{pk}_{CCR}, \boldsymbol{\pi}_{\mathbf{pk}_{CCR}}, (\mathbf{ee}, \text{"GenKeysCCR"}))$ ▷ See algorithm 3.18
 - 2: **if** $\neg \text{VerifSch}$ **then**
 - 3: **return** \perp ▷ If any proof fails, the algorithm aborts and returns nothing
 - 4: **end if**
 - 5: $\mathbf{pk}_{CCR} \leftarrow \text{CombinePublicKeys}((\mathbf{pk}_{CCR_1}, \mathbf{pk}_{CCR_2}, \mathbf{pk}_{CCR_3}, \mathbf{pk}_{CCR_4}))$ ▷ See crypto primitives specification
-

Output:

Choice Return Codes encryption public key $\mathbf{pk}_{CCR} \in \mathbb{G}_q^{\psi_{\max}}$

4.1.9 GenCredDat

The setup component generates the voter's credential data in the algorithm GenCredDat.

Algorithm 4.11 GenCredDat

Context:

- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$
- Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
- Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
- Vector of verification card IDs $\mathbf{vc} = (\mathbf{vc}_0, \dots, \mathbf{vc}_{N_E-1}) \in (\mathbb{A}_{Base16})^{1_{ID} \times N_E}$
- Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$
- Election public key $\mathbf{EL}_{pk} = (\mathbf{EL}_{pk,0}, \dots, \mathbf{EL}_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$
- Choice Return Codes encryption public key $\mathbf{pk}_{CCR} = (\mathbf{pk}_{CCR,0}, \dots, \mathbf{pk}_{CCR,\psi_{max}-1}) \in \mathbb{G}_q^{\psi_{max}}$

Input:

- Vector of verification card secret keys $\mathbf{k} = (\mathbf{k}_0, \dots, \mathbf{k}_{N_E-1}) \in \mathbb{Z}_q^{N_E}$
 - Vector of Start Voting Keys $\mathbf{SVK} = (\mathbf{SVK}_0, \dots, \mathbf{SVK}_{N_E-1}) \in ((\mathbb{A}_{u32})^{1_{SVK}})^{N_E}$
-

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $\mathbf{i}_{aux} \leftarrow (\text{"GetKey"}, \text{GetHashContext}()) \triangleright$ See algorithm 3.12
 - 2: **for** $\mathbf{id} \in [0, N_E]$ **do**
 - 3: $(\mathbf{dSVK}_{id}, \mathbf{VCks}_{id,salt}) \leftarrow \text{GenArgon2id}(\text{StringToByteArray}(\mathbf{SVK}_{id})) \triangleright$ Use the Argon2 profile for less memory, as defined in the crypto primitives specification
 - 4: $\mathbf{KSkey}_{id} \leftarrow \text{RecursiveHash}(\text{"VerificationCardKeystore"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{id}, \mathbf{dSVK}_{id})$
 - 5: $\mathbf{k}_{id,bytes} \leftarrow \text{IntegerToFixedLengthByteArray}(\mathbf{k}_{id}, \lceil \frac{|q|}{8} \rceil)$
 - 6: $(\mathbf{VCks}_{id,ciphertext}, \mathbf{VCks}_{id,nonce}) \leftarrow \text{GenCiphertextSymmetric}(\mathbf{KSkey}_{id}, \mathbf{k}_{id,bytes}, \mathbf{i}_{aux})$
 - 7: $\mathbf{VCks}_{id} \leftarrow \text{Base64Encode}(\mathbf{VCks}_{id,ciphertext} || \mathbf{VCks}_{id,nonce} || \mathbf{VCks}_{id,salt})$
 - 8: **end for**
-

Output:

- Vector of verification card keystores $\mathbf{VCks} = (\mathbf{VCks}_0, \dots, \mathbf{VCks}_{N_E-1}) \in (\mathbb{A}_{Base64}^{1_{VCks}})^{N_E}$
-

4.2 SetupTally

The SetupTally protocol sets up the key to encrypt the actual vote. SetupTally consists of the SetupTallyCCM and SetupTallyEB algorithm. The SetupTally algorithms integrate the election event context in the challenges of the zero-knowledge proofs, enabling ~~verifiers~~ the other control components and the auditors' verifier to confirm that the control components have a consistent view of the election event context ~~Figure~~ (see section 3.6.5). Figure 8 shows the relevant algorithms and data flows.

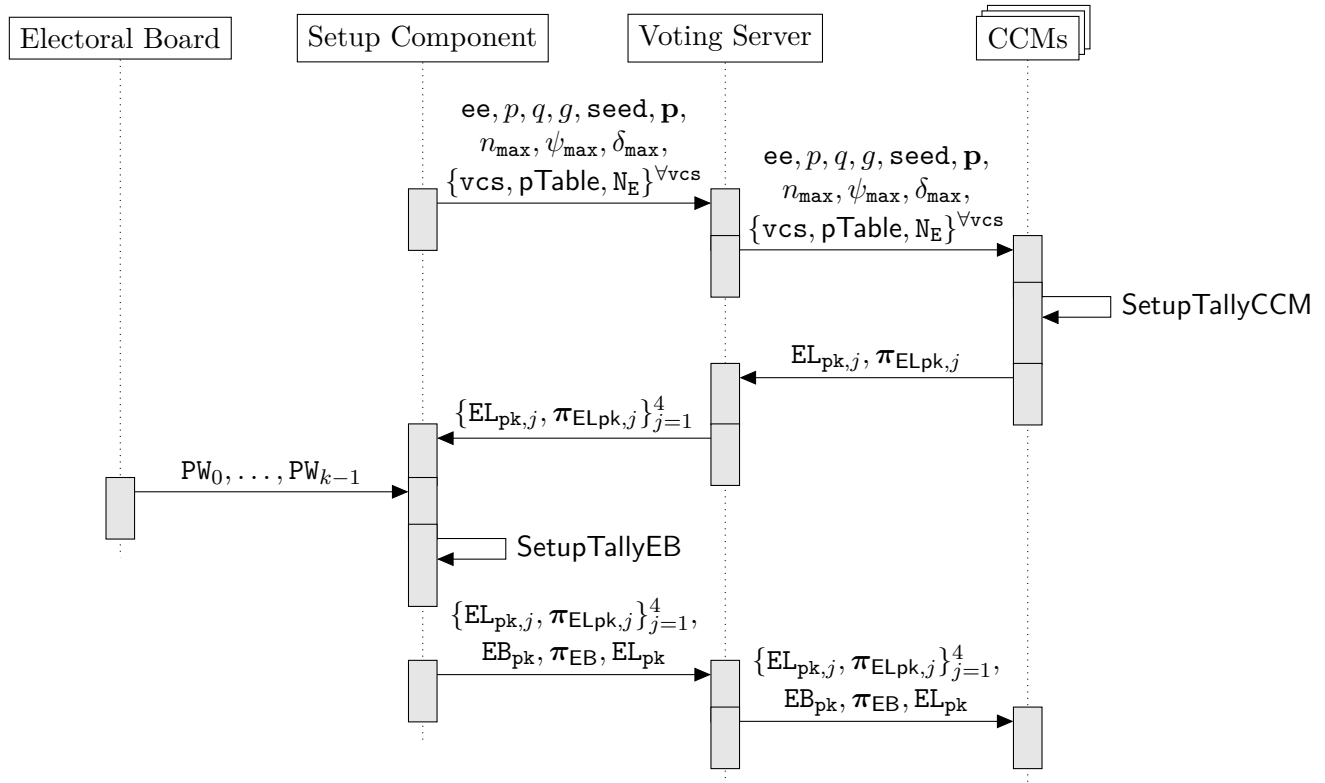


Fig. 8: Sequence diagram of the ~~SetupTally~~ SetupTally protocol.

4.2.1 SetupTallyCCM

Each Mixing control component CCM generates a key pair and proves knowledge of the secret key.

Algorithm 4.12 SetupTallyCCM

Context:

CCM's index $j \in [1, 4]$

The Election Event Context

▷ See section 3.4

▷ Including p, q, g, δ_{\max}

Operation:

▷ For all algorithms see the crypto primitives specification

1: $\text{hContext} \leftarrow \text{GetHashElectionEventContext}()$

▷ See algorithm 3.11

2: $(\text{EL}_{\text{pk},j}, \text{EL}_{\text{sk},j}) \leftarrow \text{GenKeyPair}(p, q, g, \delta_{\max})$

3: $\mathbf{i}_{\text{aux}} \leftarrow (\text{hContext}, \text{"SetupTallyCCM"}, \text{IntegerToString}(j))$

4: **for** $i \in [0, \delta_{\max})$ **do**

5: $\pi_{\text{ELpk},j,i} \leftarrow \text{GenSchnorrProof}(\text{EL}_{\text{sk},j,i}, \text{EL}_{\text{pk},j,i}, \mathbf{i}_{\text{aux}})$

6: **end for**

7: $\boldsymbol{\pi}_{\text{ELpk},j} \leftarrow (\pi_{\text{ELpk},j,0}, \dots, \pi_{\text{ELpk},j,\delta_{\max}-1})$

Output:

CCM_j election public key $\text{EL}_{\text{pk},j} \in \mathbb{G}_q^{\delta_{\max}}$

CCM_j election secret key $\text{EL}_{\text{sk},j} \in \mathbb{Z}_q^{\delta_{\max}}$

CCM_j Schnorr proofs of knowledge $\boldsymbol{\pi}_{\text{ELpk},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}}$

4.2.2 SetupTallyEB

The setup component interacts with the electoral board (see section 2.5) and derives the electoral board key pair (EB_{pk}, EB_{sk}) from the electoral board members' password, verifies the proofs of knowledge of the CCM_j election secret key $EL_{sk,j}$, and combines the CCM_j election public key $EL_{pk,j}$ and electoral board public key EB_{pk} to yield the election public key EL_{pk} . Deriving the electoral board secret key EB_{sk} from the electoral board members' passwords provides an operational safeguard against the premature decryption of the election results. As an additional operational safeguard, the setup component sends non-invertible hashes of the electoral board passwords to the Tally control component, so that the Tally control component can verify each password independently before deriving the Electoral Board secret key. The electoral board passwords must comply with a suitable password policy. For a given security strength of 128 bits, and assuming an alphabet of at least 128 characters, thus ensuring 7-bit entropy per character, a password of character length 19 is sufficient. The hashes of the passwords are generated using the Argon2id algorithm with the STANDARD profile (see crypto primitives specification).

At the end of SetupTallyEB, the setup component sends the public keys to the control components.

The control components and tally control component initialize the following lists:

- Control component's list of shuffled and decrypted ballot boxes $L_{bb,j}$
- Tally control component's list of shuffled and decrypted ballot boxes $L_{bb,Tally}$

The voting server initializes the following lists:

- Key-value map of number of authentication attempts per credential ID $L_{authAttempts}$; initially with all 0-values.
- Key-value map of the used successful authentication challenges per credential ID $L_{authChallenge}$; initially as an empty list.

Algorithm 4.13 SetupTallyEB

Context:

The Election Event Context \triangleright See section 3.4 \triangleright Including $p, q, g, \mathbf{ee}, \delta_{\max}$

Input:

CCM election public keys $\mathbf{EL}_{\text{pk}} = (\mathbf{EL}_{\text{pk},1}, \mathbf{EL}_{\text{pk},2}, \mathbf{EL}_{\text{pk},3}, \mathbf{EL}_{\text{pk},4}) \in (\mathbb{G}_q^{\delta_{\max}})^4$

CCM Schnorr proofs of knowledge $\boldsymbol{\pi}_{\text{ELpk}} = (\boldsymbol{\pi}_{\text{ELpk},1}, \boldsymbol{\pi}_{\text{ELpk},2}, \boldsymbol{\pi}_{\text{ELpk},3}, \boldsymbol{\pi}_{\text{ELpk},4}) \in ((\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}})^4$

Passwords of k electoral board members $(\text{PW}_0, \dots, \text{PW}_{k-1}) \in (\mathbb{A}_{UCS}^*)^k \triangleright$ We expect the passwords to fulfill a suitable policy

Require: $k \geq 2$

\triangleright We require at least 2 electoral board members

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $\text{hContext} \leftarrow \text{GetHashElectionEventContext}()$ \triangleright See algorithm 3.11
- 2: $\mathbf{i}_{\text{aux,CCM}} \leftarrow (\text{hContext}, \text{"SetupTallyCCM"})$
- 3: $\text{VerifSch} \leftarrow \text{VerifyCCSchnorrProofs}(\mathbf{EL}_{\text{pk}}, \boldsymbol{\pi}_{\text{ELpk}}, \mathbf{i}_{\text{aux,CCM}})$ \triangleright See algorithm 3.18
- 4: **if** $\neg \text{VerifSch}$ **then**
- 5: **return** \perp \triangleright If any proof fails, the algorithm aborts and returns nothing
- 6: **end if**
- 7: $\mathbf{i}_{\text{aux,EB}} \leftarrow (\text{hContext}, \text{"SetupTallyEB"}, \text{IntegerToString}(1))$ \triangleright Dummy value 1 is added for consistency with other proofs
- 8: **for** $i \in [0, \delta_{\max})$ **do**
- 9: $\text{EB}_{\text{sk},i} \leftarrow \text{RecursiveHashToZq}(q, (\text{"ElectoralBoardSecretKey"}, \mathbf{ee}, i, \text{PW}_0, \dots, \text{PW}_{k-1}))$
- 10: $\text{EB}_{\text{pk},i} \leftarrow g^{\text{EB}_{\text{sk},i}} \bmod p$
- 11: $\boldsymbol{\pi}_{\text{EB},i} \leftarrow \text{GenSchnorrProof}(\text{EB}_{\text{sk},i}, \text{EB}_{\text{pk},i}, \mathbf{i}_{\text{aux,EB}})$
- 12: **end for**
- 13: $\text{EB}_{\text{pk}} \leftarrow (\text{EB}_{\text{pk},0}, \dots, \text{EB}_{\text{pk},\delta_{\max}-1})$
- 14: $\boldsymbol{\pi}_{\text{EB}} \leftarrow (\boldsymbol{\pi}_{\text{EB},0}, \dots, \boldsymbol{\pi}_{\text{EB},\delta_{\max}-1})$
- 15: $\mathbf{EL}_{\text{pk}} \leftarrow \text{CombinePublicKeys}((\mathbf{EL}_{\text{pk},1}, \mathbf{EL}_{\text{pk},2}, \mathbf{EL}_{\text{pk},3}, \mathbf{EL}_{\text{pk},4}, \text{EB}_{\text{pk}}))$

Output:

Election public key $\mathbf{EL}_{\text{pk}} = (\mathbf{EL}_{\text{pk},0}, \dots, \mathbf{EL}_{\text{pk},\delta_{\max}-1}) \in \mathbb{G}_q^{\delta_{\max}}$

Electoral board public key $\text{EB}_{\text{pk}} = (\text{EB}_{\text{pk},0}, \dots, \text{EB}_{\text{pk},\delta_{\max}-1}) \in \mathbb{G}_q^{\delta_{\max}}$

Electoral board Schnorr proofs of knowledge $\boldsymbol{\pi}_{\text{EB}} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\max}}$

At the end of **SetupTallyEB**, the setup-component sends the public keys and the election event context (see section 3.4), including the voting options configuration (see section 3.5) to the control components, the Tally control component, and the auditors (see figure 7 and 8).

4.3 Finalize Configuration Phase

The control components verify Figure 9 shows the different data flows following the **SetupVoting** and **SetupTally** protocols.

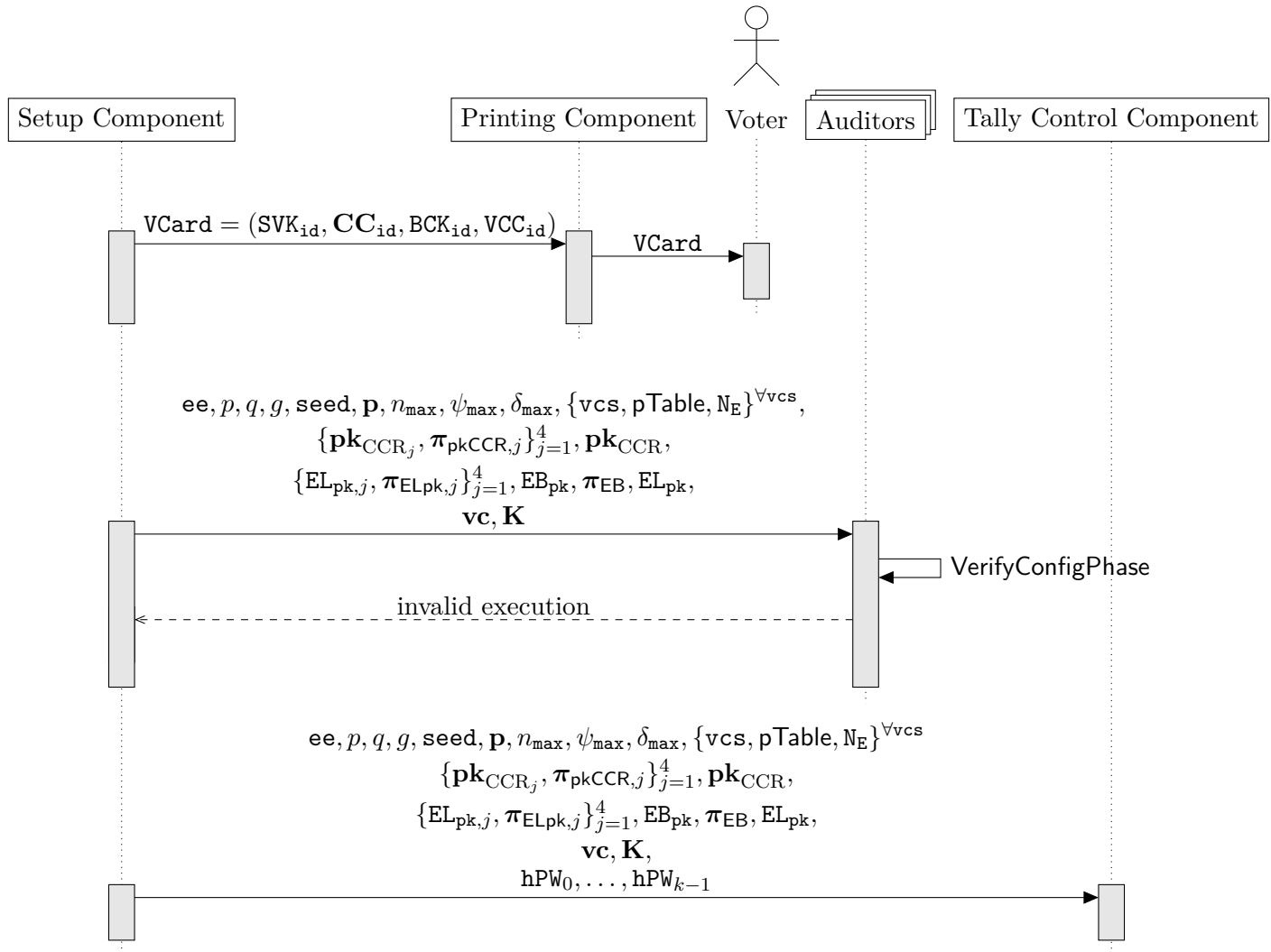


Fig. 9: Sequence diagram of the finalization of the configuration phase.

From a protocol perspective, executing **VerifyConfigPhase** during the configuration phase is not strictly required, as it would be sufficient to run it at the end of the tally phase. However, performing the verification earlier provides participants with assurance that the election event context is consistent with the information processed in **GenEncLongCodeShares**. In particular, configuration is correct before the voting phase starts. It also allows the verification to be combined with other ceremonial tasks of the auditors, such as submitting test votes in test ballot boxes.

After the control components ~~check~~ have received the final data flows shown in figures 7 and 8, they ensure the following:

- ~~The number of verification cards for each verification card set corresponds to the number of verification cards processed in `GenEncLongCodeShares`~~Each control component verifies the other components' proofs of correct key generation and confirms that all parties implicitly agree on the election event context (see Section 3.6.5).
- ~~The correctness of the setup and other control components' key generation by executing the `VerifyKeyGenerationSchnorrProofs` algorithm.~~

~~Moreover, the control components and tally control component initialize the following lists:—~~

- ~~Control component's list of shuffled and decrypted ballot boxes $L_{bb,j}$~~
- ~~Tally control component's list of shuffled and decrypted ballot boxes $L_{bb,TALLY}$~~

~~The voting server initializes the following lists:—~~

- ~~Key-value map of number of authentication attempts per credential ID $L_{authAttempts}$; initially with all 0-values.~~have received all election-event-specific data from the setup component. During the voting phase, they will explicitly agree on this data (see Section 3.6.3).
- ~~Key-value map of the used successful authentication challenges per credential ID $L_{authChallenge}$; initially as an empty list~~The control components block the execution of the configuration phase algorithms listed in table 11 and enable the execution of the voting phase algorithms once the start time of the ballot boxes is reached.

5 Voting Phase

The voting phase consists of three sub-protocols: AuthenticateVoter, SendVote and ConfirmVote. Table 12 shows the algorithms involved.

Algorithm	Actor	Reference
AuthenticateVoter		Figure 10
GetAuthenticationChallenge	Voting Client	5.1
VerifyAuthenticationChallenge	Voting Server	5.2
GetKey	Voting Client	5.3
SendVote		Figure 11
CreateVote	Voting Client	5.4
VerifyBallotCCR	Control Component (CCR)	5.5
PartialDecryptPCC	Control Component (CCR)	5.6
DecryptPCC	Control Component (CCR)	5.7
CreateLCCShare	Control Component (CCR)	5.8
ExtractCRC	Voting Server	5.9
ConfirmVote		Figure 12
CreateConfirmMessage	Voting Client	5.10
CreateLVCCShare	Control Component (CCR)	5.11
VerifyLVCCHash	Control Component (CCR)	5.12
ExtractVCC	Voting Server	5.13

Tab. 12: Overview of the algorithms in the voting phase.

To access the voting client application, the voter enters a URL that is printed on their voting card. This URL directs the voter to a website that contains a link to start the voting process for a specific election event. The link includes the election event ID and directs the voter to the actual voting client application.

The annex of the Federal Chancellery's Ordinance requires that the voter can check the correctness of the voting client.

[VEleS Annex 2.7.3]: It must be ensured that no attacker can take control of user devices unnoticed by manipulating the user device software on the server. The person voting must be able to verify that the server has provided his or her user device with the correct software with the correct parameters, in particular the public key for encrypting the vote.

The Swiss Post Voting System fulfills this requirement by publishing the hashes of the voting client via an out-of-band channel. Thereby, a voter can check that her voting client is correct. A correct voting client checks the correctness of the public key in the **GetKey** algorithm (see section 5.1.3).

5.1 AuthenticateVoter

In the AuthenticateVoter phase, the voter enters the Start Voting Key SVK_{id} and an extended authentication factor EA_{id} to the voting client. (For further description of the extended authentication factor, see section 4.1.3.) The voting client derives the base authentication challenge $hAuth_{id}$ from SVK_{id} and EA_{id} and authenticates to the voting server.

At the end of the AuthenticateVoter phase, the voting client ~~receives~~receives the election event context, including all relevant public keys, and ~~extracted~~extracts the verification card secret key k_{id} .

Subsequently, the voting client displays the voting options to the voter, who selects the desired voting options prior to the SendVote phase.

Although the voting server is considered untrusted for the purpose of verifiability and voting secrecy, performing an authentication protocol between the voting client and the voting server increases the robustness of the system. The voting server acts as an intermediary layer that provides a first line of defense against malformed or malicious requests. In practice, it helps to prevent certain types of attacks, such as distributed denial-of-service attacks or attempts to drive the control components into an inconsistent state, by blocking invalid authentication attempts early. While this does not strengthen the cryptographic guarantees of verifiability or voting secrecy, it contributes to the overall operational stability and integrity of the system by shielding the control components.

Throughout the voting phase, the voting client and voting server execute the `GetAuthenticationChallenge` and `VerifyAuthenticationChallenge` algorithms at each interaction, including the SendVote and ConfirmVote phases. This ensures the freshness of the voting client's request and proves knowledge of the base authentication challenge $hAuth_{id}$.

The AuthenticateVoter phase is heavily inspired by the Time-based One-Time Password (TOTP) protocol specified in RFC6238, a widely-used authentication method with robust security proofs [16]. TOTP's timestamp component is used to prevent replay attacks. The shared secret between voting client and voting server is the base authentication challenge $hAuth_{id}$. The voting server ~~receives~~receives the base authentication challenge $hAuth_{id}$ from the setup component while the voting client derives it from SVK_{id} and EA_{id} ~~receives~~received from the voter. RFC6238 recommends that one backward time step is allowed as the network delay, and that the validator tolerates the prover being "out of sync" by a specific number of backward and forward time steps before being rejected. We allow one forward and one backward time step. However, in some aspects, we deviate from RFC6238. First, an authentication **nonce** ensures that multiple authentication attempts within the same time step are possible. Second, while RFC6238 recommends a time step size of 30 seconds, we use a time step size of 300 seconds. Because of the usage of the **nonce**, allowing further time windows (as recommended by RFC6238) has the same result as increasing the time step size. The latter approach is preferred because of the better performance. Third, we use Argon2id [3], a state-of-the-art memory-hard key derivation function, instead of HMAC-SHA-256. This choice ensures consistency with the use of Argon2id in the voting client while keeping the authentication process secure.

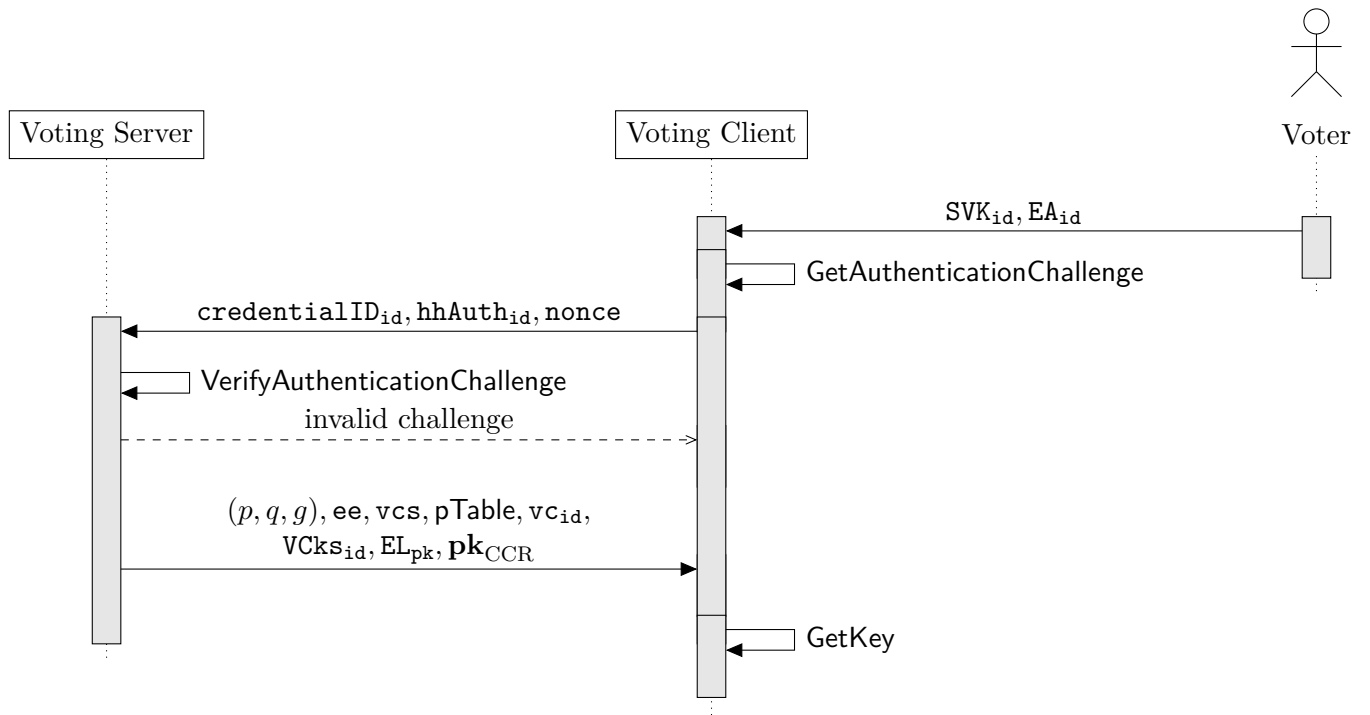


Fig. 10: Sequence diagram of the AuthenticateVoter sub-protocol

5.1.1 GetAuthenticationChallenge

The voting client executes the **GetAuthenticationChallenge** algorithm multiple times during the voting process. To distinguish different invocations of the **GetAuthenticationChallenge** algorithm, we include the corresponding authentication step (**authenticateVoter**, **sendVote**, **confirmVote**) in the salt.

Since authentication might happen multiple times within a given time step, the voting client generates an authentication **nonce** for every authentication attempt—allowing the voting client and voting server to distinguish different authentication attempts within the same time step. Subsequently, the voting client sends the authentication **nonce** to the voting server.

Algorithm 5.1 GetAuthenticationChallenge

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Character length of the extended authentication factor $1_{EA} \in \mathbb{N}^+$

Input:

Authentication step $authStep \in (\text{"authenticateVoter"}, \text{"sendVote"}, \text{"confirmVote"})$

Start Voting Key $SVK_{id} \in (\mathbb{A}_{u32})^{1_{SVK}}$

Extended authentication factor $EA_{id} \in \mathbb{A}_{10}^{1_{EA}}$

Operation:

▷ For all algorithms see the crypto primitives specification

- 1: $credentialID_{id} \leftarrow \text{DeriveCredentialId}(SVK_{id})$ ▷ See algorithm 3.19
 - 2: $hAuth_{id} \leftarrow \text{DeriveBaseAuthenticationChallenge}(SVK_{id}, EA_{id})$ ▷ See algorithm 3.20
 - 3: $nonce \leftarrow \text{GenRandomInteger}(2^{256})$
 - 4: $TS \leftarrow \text{GetTimestamp}()$ ▷ Returns the current Unix time, measured in number of seconds passed since 01.01.1970
 - 5: $T \leftarrow \lfloor \frac{TS}{300} \rfloor$
 - 6: $salt_{id} \leftarrow \text{CutToBitLength}(\text{RecursiveHash}(ee, credentialID_{id}, \text{"dAuth"}, authStep, nonce), 128)$
 - 7: $k \leftarrow (\text{StringToByteArray}(hAuth_{id}) \parallel \text{StringToByteArray}(\text{"Auth"}) \parallel \text{IntegerToByteArray}(T))$
 - 8: $bhhAuth_{id} \leftarrow \text{GetArgon2id}(k, salt_{id})$ ▷ Use the Argon2 profile for less memory, as defined in the crypto primitives specification
 - 9: $hhAuth_{id} \leftarrow \text{Base64Encode}(bhhAuth_{id})$
-

Output:

Derived voter identifier $credentialID_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Derived authentication challenge $hhAuth_{id} \in (\mathbb{A}_{Base64})^{1_{HB64}}$

Authentication **nonce** $\in [0, 2^{256})$

5.1.2 VerifyAuthenticationChallenge

After receiving the authentication challenge from the voting client, the voting server must verify its validity. Table 13 shows the source of the context and input variables and the preliminary validations that the voting server performs before invoking the algorithm.

Information	Variable	Source	Use as	Preliminary Validation
Election event ID	ee	Voting Client	Context	Check that ee exists in the internal view.
Derived voter identifier	credentialID_{id}	Voting Client	Context	Check in the internal view that credentialID_{id} corresponds to a vc_{id} for this ee , that the corresponding ballot box is currently open, and that the vc_{id} is consistent with other information received from the voting client.
Authentication step	authStep	Voting Client	Input	Check that the authentication step is consistent with the state of the vc_{id} .
Derived authentication challenge	hhAuth_{id}	Voting Client	Input	None. Checked within algorithm 5.2.
Base authentication challenge	hAuth_{id}	Internal view	Input	None, since retrieved from the trusted internal view.
Authentication nonce	nonce	Voting Client	Input	None, other than the implicit domain checks.

Tab. 13: Context and input validation for VerifyAuthenticationChallenge

Moreover, the voting server must keep track of the number of authentication attempts made by the voter. The voting server maintains the following two stateful maps:

- Key-value map of number of authentication attempts per credential ID $L_{\text{authAttempts}}$
- Key-value map of the used authentication challenges per credential ID $L_{\text{authChallenge}}$

The stateful lists prevent a voting client from brute forcing the extended authentication factor and enforce one-time use of an authentication challenge as specified in RFC6238 [16].

Algorithm 5.2 VerifyAuthenticationChallenge

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1ID}$

Derived voter identifier $credentialID_{id} \in (\mathbb{A}_{Base16})^{1ID}$

Stateful Lists and Maps:

Key-value map of number of authentication attempts per credential ID $L_{authAttempts}$

Key-value map of the used successful authentication challenges per credential ID $L_{authChallenge}$

Input:

Authentication step $authStep \in (\text{"authenticateVoter"}, \text{"sendVote"}, \text{"confirmVote"})$

Derived authentication challenge $hhAuth_{id} \in (\mathbb{A}_{Base64})^{1HB64}$

Base authentication challenge $hAuth_{id} \in (\mathbb{A}_{Base64})^{1HB64}$

Authentication nonce $\in [0, 2^{256})$

Operation:

▷ For all algorithms see the crypto primitives specification

```

1:  $TS \leftarrow \text{GetTimestamp}()$                                 ▷ Returns the current Unix time
2:  $T_1 \leftarrow \lfloor \frac{TS}{300} \rfloor$ 
3:  $T_0 \leftarrow T_1 - 1$ 
4:  $T_2 \leftarrow T_1 + 1$ 
5:  $attempts_{id} \leftarrow L_{authAttempts}(credentialID_{id})$  ▷ Retrieve the value from the key-value map
6: if  $((attempts_{id} \geq 5) \vee (hhAuth_{id} \in L_{authChallenge}(credentialID_{id})))$  then
    return  $\perp$                                 ▷ Enforces one-time usage and limits the voting client to 5 attempts
7: end if
8:  $salt_{id} \leftarrow \text{CutToBitLength}(\text{RecursiveHash}(ee, credentialID_{id}, \text{"dAuth"}, authStep, nonce), 128)$ 
9: for  $T_i \in \{T_1, T_0, T_2\}$  do
10:     $k \leftarrow (\text{StringToByteArray}(hAuth_{id}) \parallel \text{StringToByteArray}(\text{"Auth"}) \parallel \text{IntegerToByteArray}(T_i))$ 
11:     $bhhAuth'_{id,i} \leftarrow \text{GetArgon2id}(k, salt_{id})$  ▷ Use the Argon2 profile for less memory, as
    defined in the crypto primitives specification
12:     $hhAuth'_{id,i} \leftarrow \text{Base64Encode}(bhhAuth'_{id,i})$ 
13:    if  $(hhAuth'_{id,i} = hhAuth_{id})$  then
14:         $L_{authChallenge}(credentialID_{id}) \leftarrow L_{authChallenge}(credentialID_{id}) \cup hhAuth_{id}$ 
         $L_{authChallenge}(credentialID_{id}) \leftarrow L_{authChallenge}(credentialID_{id}) \parallel hhAuth_{id}$  ▷ Add the
        authentication challenge to the list of used challenges
15:        return  $\top$ 
16:    end if
17: end for
18:  $L_{authAttempts}(credentialID_{id}) \leftarrow attempts_{id} + 1$  ▷ Set the value in the key-value map
19: return  $\perp$ 

```

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

To prevent the list $L_{authChallenge}(credentialID_{id})$ of successful authentication attempts from becoming too long, the voting server can reinitialize the list if the time step T_0 of the current authentication attempt exceeds the time step $T_{2,last}$ of the last successful authentication. The

voting server sends the Verification Card Keystore \mathbf{VCks}_{id} to the voting client after a successful verification of the authentication challenge.

5.1.3 GetKey

In the **GetKey** algorithm, the voting client opens the Verification Card Keystore \mathbf{VCks}_{id} to obtain the verification card secret key \mathbf{k}_{id} . The voting client receives the Start Voting Key \mathbf{SVK}_{id} from the voter and the other input and context arguments from the voting server. The voting client verifies the context data's authenticity using the authenticated encryption scheme (see the section *Symmetric Authenticated Encryption* in the crypto primitives specification) and the Start Voting Key \mathbf{SVK}_{id} —a secret shared between the setup component and the voting client.

Algorithm 5.3 GetKey

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
Verification card ID $\mathbf{vc}_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$
Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$
Election public key $\mathbf{EL}_{pk} = (\mathbf{EL}_{pk,0}, \dots, \mathbf{EL}_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$
Choice Return Codes encryption public key $\mathbf{pk}_{CCR} = (\mathbf{pk}_{CCR,0}, \dots, \mathbf{pk}_{CCR,\psi_{max}-1}) \in \mathbb{G}_q^{\psi_{max}}$

Input:

Start Voting Key $\mathbf{SVK}_{id} \in (\mathbb{A}_{u32})^{1_{SVK}}$
Verification Card Keystore $\mathbf{VCks}_{id} \in \mathbb{A}_{Base64}^{1_{VCks}}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $\mathbf{i}_{aux} \leftarrow (\text{"GetKey"}, \text{GetHashContext}())$ \triangleright See algorithm 3.12
- 2: $\mathbf{VCks}_{id,combined} \leftarrow \text{Base64Decode}(\mathbf{VCks}_{id})$ $\triangleright \mathbf{VCks}_{id,combined}$ consists of
 $\mathbf{VCks}_{id,ciphertext} || \mathbf{VCks}_{id,nonce} || \mathbf{VCks}_{id,salt}$
- 3: $\mathbf{length} \leftarrow \text{Length in bytes of } \mathbf{VCks}_{id,combined}$
- 4: $\mathbf{split}_{ciphertext} \leftarrow \mathbf{length} - (\mathbf{nonceLength} + \mathbf{saltLength})$ \triangleright Nonce length in bytes defined by the symmetric algorithm used, salt length in bytes specified in section *Argon2* in the crypto primitives specification
- 5: $\mathbf{split}_{nonce} \leftarrow \mathbf{split}_{ciphertext} + \mathbf{nonceLength}$
- 6: $\mathbf{VCks}_{id,ciphertext} \leftarrow \mathbf{VCks}_{id,combined}[0 : \mathbf{split}_{ciphertext}]$
- 7: $\mathbf{VCks}_{id,nonce} \leftarrow \mathbf{VCks}_{id,combined}[\mathbf{split}_{ciphertext} : \mathbf{split}_{nonce}]$
- 8: $\mathbf{VCks}_{id,salt} \leftarrow \mathbf{VCks}_{id,combined}[\mathbf{split}_{nonce} : \mathbf{length}]$
- 9: $\mathbf{dSVK}_{id} \leftarrow \text{GetArgon2id}(\text{StringToByteArray}(\mathbf{SVK}_{id}), \mathbf{VCks}_{id,salt})$ \triangleright Use the Argon2 profile for less memory, as defined in the crypto primitives specification
- 10: $\mathbf{KSkey}_{id} \leftarrow \text{RecursiveHash}(\text{"VerificationCardKeystore"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{id}, \mathbf{dSVK}_{id})$
- 11: $\mathbf{k}_{id,bytes} \leftarrow \text{GetPlaintextSymmetric}(\mathbf{KSkey}_{id}, \mathbf{VCks}_{id,ciphertext}, \mathbf{VCks}_{id,nonce}, \mathbf{i}_{aux})$
- 12: $\mathbf{k}_{id} \leftarrow \text{ByteArrayToInteger}(\mathbf{k}_{id,bytes})$

Output:

Verification Card Secret Key $\mathbf{k}_{id} \in \mathbb{Z}_q$

5.2 SendVote

The SendVote phase encompasses creating the vote until the voter receives the short Choice Return Codes. Figure 11 shows the relevant algorithms.

SendVote assumes that the voter already authenticated to the voting server and that the voting client knows the verification card secret key \mathbf{k}_{id} and validated the election event's public keys.

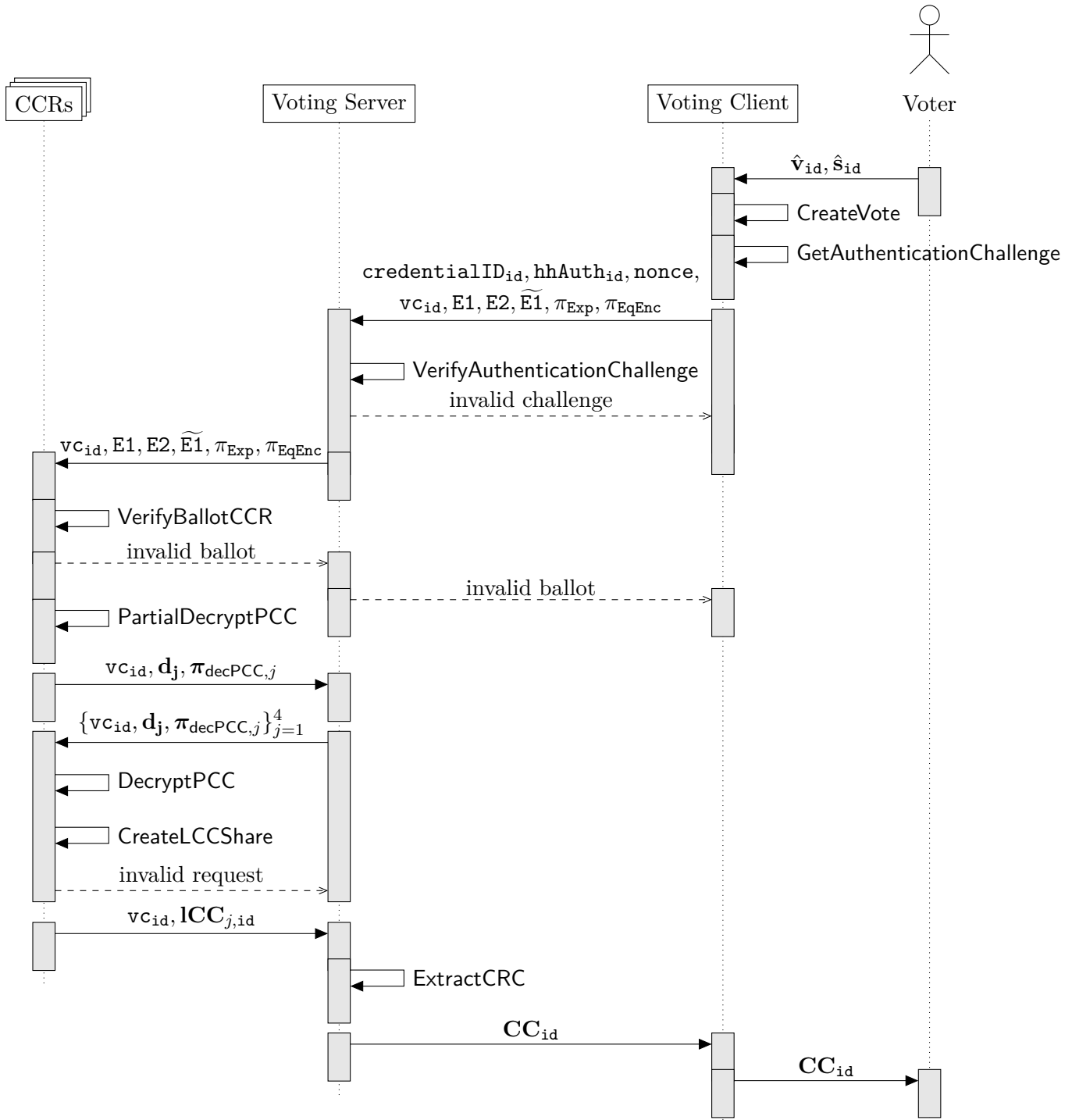


Fig. 11: Sequence diagram of the SendVote sub-protocol

5.2.1 CreateVote

The voting client creates the vote containing the encrypted voting options, the partial Choice Return Codes, and the non-interactive zero-knowledge proofs linking them. The voting client is supplied with a list of selected actual voting options $\hat{\mathbf{v}}_{\text{id}}$ and a list of selected write-ins $\hat{\mathbf{s}}_{\text{id}}$. Section 3.8.4 elaborates on the fact that the voter might select k out of $\delta - 1$ write-in options, if write-ins are permitted. The Swiss Post voting system supports encrypting write-in votes along with the pre-defined voting options. The voting client generates two ciphertexts: the first ciphertext E1 contains the encrypted voting options and—optionally—the encrypted write-ins, the second ciphertext E2 contains the basis for generating the short Choice Return Codes. Both ciphertexts are linked through non-interactive zero-knowledge proofs. A Schnorr proof π_{sch} —which proves knowledge of the encryption randomness $r \in \mathbb{Z}_q$ —is redundant:

- π_{exp} proves knowledge of the pre-image $\cancel{k_{id}} \cdot k_{id}$ (the verification card secret key) and that the prover raised the bases (g, E1) to the same exponent.
- π_{EqEnc} proves knowledge of the pre-images $\cancel{r} \cdot \cancel{k_{id}} \cdot r \cdot k_{id}$ and r' and that $\widetilde{E1}$ and $\widetilde{E2}$ contain the same message (encrypted under different public keys).

If the prover knows $\cancel{r} \cdot \cancel{k_{id}} \cdot r \cdot k_{id}$ (from the plaintext equality proof) and $\cancel{k_{id}} \cdot k_{id}$ (from the exponentiation proof), he implicitly knows r (which is the statement that the Schnorr proof claims).

Algorithm 5.4 CreateVote

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{\text{ID}}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{\text{ID}}}$
 Verification card ID $\mathbf{vc}_{\text{id}} \in (\mathbb{A}_{Base16})^{1_{\text{ID}}}$
 Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ \triangleright \mathbf{pTable} is of the form
 $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ \triangleright ψ and δ can be derived from \mathbf{pTable} using algorithms 3.8 and 3.9
 Election public key $\mathbf{EL}_{\text{pk}} = (\mathbf{EL}_{\text{pk},0}, \dots, \mathbf{EL}_{\text{pk},\delta_{\text{max}}-1}) \in \mathbb{G}_q^{\delta_{\text{max}}}$
 Choice Return Codes encryption public key $\mathbf{pk}_{\text{CCR}} \in \mathbb{G}_q^{\psi_{\text{max}}}$

Input:

Selected actual voting options $\hat{\mathbf{v}}_{\text{id}} = (\hat{v}_0, \dots, \hat{v}_{\psi-1}) \in (\mathcal{T}_1^{50})^\psi$ \triangleright See section 3.5
 Selected write-ins $\hat{\mathbf{s}}_{\text{id}} = (\hat{s}_0, \dots, \hat{s}_{k-1}) \in ((\mathbb{A}_{\text{latin}} \setminus \#)^*)^k$ \triangleright See section 3.8
 Verification card secret key $\mathbf{k}_{\text{id}} \in \mathbb{Z}_q$

Require: $\text{GetBlankCorrectnessInformation}() = \text{GetCorrectnessInformation}(\hat{\mathbf{v}}_{\text{id}})$ \triangleright See algorithms 3.5 and 3.6.

The algorithm 3.5 ensures $\hat{\mathbf{v}}_{\text{id}}$ is a subset of $\tilde{\mathbf{v}}$ and contains no duplicates.

Require: $k \leq \delta - 1$ \triangleright $\delta = 1$, if the ballot box does not have any write-in candidates.

Require: $|\hat{s}_i| < 1_w, \forall i \in [0, k)$ \triangleright where $|\hat{s}_i|$ is the character length of \hat{s}_i

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $(\hat{p}_0, \dots, \hat{p}_{\psi-1}) \leftarrow \text{GetEncodedVotingOptions}(\hat{\mathbf{v}}_{\text{id}})$ \triangleright See algorithm 3.2
- 2: $(w_{\text{id},0}, \dots, w_{\text{id},\delta-2}) \leftarrow \text{EncodeWriteIns}(\hat{\mathbf{s}}_{\text{id}})$ \triangleright See algorithm 3.25
- 3: $\rho \leftarrow \prod_{i=0}^{\psi-1} \hat{p}_i \bmod p$
- 4: $r \leftarrow \text{GenRandomInteger}(q)$
- 5: $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \leftarrow \text{GetCiphertext}((\rho, w_{\text{id},0}, \dots, w_{\text{id},\delta-2}), r, \mathbf{EL}_{\text{pk}})$
- 6: **for** $i \in [0, \psi)$ **do**
- 7: $\mathbf{pCC}_{\text{id},i} \leftarrow \hat{p}_i^{k_{\text{id}}} \bmod p$
- 8: **end for**
- 9: $\mathbf{pCC}_{\text{id}} = (\mathbf{pCC}_{\text{id},0}, \dots, \mathbf{pCC}_{\text{id},\psi-1})$
- 10: $r' \leftarrow \text{GenRandomInteger}(q)$
- 11: $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \leftarrow \text{GetCiphertext}(\mathbf{pCC}_{\text{id}}, r', \mathbf{pk}_{\text{CCR}})$
- 12: $\widetilde{\mathbf{E1}} \leftarrow \text{GetCiphertextExponentiation}((\gamma_1, \phi_{1,0}), \mathbf{k}_{\text{id}})$
- 13: $\widetilde{\mathbf{E2}} \leftarrow (\gamma_2, \prod_{i=0}^{\psi-1} \phi_{2,i} \bmod p)$
- 14: $K_{\text{id}} \leftarrow g^{k_{\text{id}}} \bmod p$
- 15: $\mathbf{i}_{\text{aux}} \leftarrow (\text{"CreateVote"}, \mathbf{vc}_{\text{id}}, \text{GetHashContext}())$ \triangleright See algorithm 3.12
- 16: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$
- 17: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}), \dots, \text{IntegerToString}(\phi_{2,\psi-1}))$
- 18: $\pi_{\text{Exp}} \leftarrow \text{GenExponentiationProof}((g, \gamma_1, \phi_{1,0}), \mathbf{k}_{\text{id}}, (K_{\text{id}}, \gamma_1^{k_{\text{id}}}, \phi_{1,0}^{k_{\text{id}}}), \mathbf{i}_{\text{aux}})$
- 19: $\widetilde{\mathbf{pk}}_{\text{CCR}} \leftarrow \prod_{i=0}^{\psi-1} \mathbf{pk}_{\text{CCR},i} \bmod p$
- 20: $\pi_{\text{EqEnc}} \leftarrow \text{GenPlaintextEqualityProof}(\widetilde{\mathbf{E1}}, \widetilde{\mathbf{E2}}, \mathbf{EL}_{\text{pk},0}, \widetilde{\mathbf{pk}}_{\text{CCR}}, (r \cdot K_{\text{id}}, r'), \mathbf{i}_{\text{aux}})$

Output:

Encrypted vote $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$
 Encrypted partial Choice Return Codes $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$
 Exponentiated encrypted vote $\widetilde{\mathbf{E1}} \in \mathbb{G}_q^2$
 Exponentiation proof $\pi_{\text{Exp}} \in \mathbb{Z}_q \times \mathbb{Z}_q$
 Plaintext equality proof $\pi_{\text{EqEnc}} \in \mathbb{Z}_q \times \mathbb{Z}_q^2$

5.2.2 VerifyBallotCCR

The Return Codes control components CCR check the voting client's encrypted vote, in particular the zero-knowledge proofs.

Algorithm 5.5 VerifyBallotCCR

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card ID $\mathbf{vc}_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ \triangleright \mathbf{pTable} is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ \triangleright ψ and δ can be derived from \mathbf{pTable} using algorithms 3.8 and 3.9
 Verification card public key $\mathbf{K}_{id} \in \mathbb{G}_q$
 Election public key $\mathbf{EL}_{pk} = (\mathbf{EL}_{pk,0}, \dots, \mathbf{EL}_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$
 Choice Return Codes encryption public key $\mathbf{pk}_{CCR} \in \mathbb{G}_q^{\psi_{max}}$

Input:

Encrypted vote $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$
 Exponentiated encrypted vote $\widetilde{\mathbf{E1}} = (\gamma_1^{k_{id}}, \phi_{1,0}^{k_{id}}) \in \mathbb{G}_q^2$
 Encrypted partial Choice Return Codes $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$
 Exponentiation proof $\pi_{Exp} \in \mathbb{Z}_q \times \mathbb{Z}_q$
 Plaintext equality proof $\pi_{EqEnc} \in \mathbb{Z}_q \times \mathbb{Z}_q^2$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1:  $\widetilde{\mathbf{E2}} \leftarrow (\gamma_2, \prod_{i=0}^{\psi-1} \phi_{2,i} \bmod p)$ 
2:  $\widetilde{\mathbf{pk}}_{CCR} \leftarrow \prod_{i=0}^{\psi-1} \mathbf{pk}_{CCR,i} \bmod p$ 
3:  $\mathbf{i}_{aux} \leftarrow (\text{"CreateVote"}, \mathbf{vc}_{id}, \text{GetHashContext}())$   $\triangleright$  See algorithm 3.12
4:  $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$ 
5:  $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}), \dots, \text{IntegerToString}(\phi_{2,\psi-1}))$ 
6:  $\text{VerifExp} \leftarrow \text{VerifyExponentiation}((g, \gamma_1, \phi_{1,0}), (\mathbf{K}_{id}, \gamma_1^{k_{id}}, \phi_{1,0}^{k_{id}}), \pi_{Exp}, \mathbf{i}_{aux})$ 
7:  $\text{VerifEqEnc} \leftarrow \text{VerifyPlaintextEquality}(\widetilde{\mathbf{E1}}, \widetilde{\mathbf{E2}}, \mathbf{EL}_{pk,0}, \widetilde{\mathbf{pk}}_{CCR}, \pi_{EqEnc}, \mathbf{i}_{aux})$ 
8: if  $\text{VerifExp} \wedge \text{VerifEqEnc}$  then
    return  $\top$ 
9: else
    return  $\perp$ 
10: end if
```

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

The VerifyBallotCCR algorithm does *not* modify any state. However, the subsequent algorithm requires the successful execution of *all* verifications and checks.

5.2.3 PartialDecryptPCC

~~Provided that~~ Upon successful execution of VerifyBallotCCR ~~is successfully executed,~~ the control components ~~strip the partial Choice Return Codes's encryption layer . remove the encryption layer from the partial Choice Return Codes~~ pCC_{id} .

One might ask why ~~we encrypt the partial Choice Return Codes~~ pCC_{id} ~~even though~~ pCC_{id} ~~is encrypted at all, given that both~~ the untrustworthy voting client and voting server learn ~~these codes' plaintexts~~ the plaintext codes in any case during normal protocol execution. ~~For historical reasons~~ Historically, the protocol ~~defended~~ included this encryption layer to protect against eavesdroppers trying to infer whether the voter chose the same voting option twice; exponentiating the same prime number to the same verification card secret key k_{id} results in the same partial Choice Return ~~Codes~~ Code. Currently, both conditions no longer hold: ~~our~~ the current protocol defends against a much stronger adversary (controlling the voting server), and ~~we enforce~~ enforces that the voting client cannot select the same voting option twice. ~~Nevertheless, we maintained~~

~~Nevertheless,~~ the additional encryption layer remains in the current ~~version of the protocol.~~ ~~This algorithm~~ protocol version since it is still relevant for other parts of the system. The encryption layer, and in particular the partial decryption executed by the control components, plays an important role for the agreement procedure before tallying to resolve potential discrepancies (see section 6.2). In the partial decryption step, each control component computes an exponentiation proof. This exponentiation proof acts as a commitment to the content of the encrypted vote. Before processing the encrypted vote, each control component verifies the exponentiation proofs from all other control components. Hence, providing a set of one valid exponentiation proof for each control component confirms that they all committed to process this vote. This ensures that for each voter, there is consensus on which encrypted vote to be included in the tally.

[Algorithm 5.6](#) uses the list of voting cards $L_{\text{decPCC},j}$ for which the control components decrypted the partial Choice Return Codes \mathbf{pCC}_{id} .

Algorithm 5.6 PartialDecryptPCC

Context:

~~Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$~~ The CCR's index $j \in [1, 4]$
~~Election event ID $\mathbf{ee} \in (\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}}$ Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}}$~~ Verification card ID $\mathbf{vc}_{\text{id}} \in (\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}}$
Number of allowed selections for this specific ballot box $\psi \in [1, \psi_{\text{sup}}]$ \triangleright Can be derived from \mathbf{pTable} using algorithm 3.8
~~Election public key $\mathbf{EL}_{\text{pk}} = (\mathbf{EL}_{\text{pk},0}, \dots, \mathbf{EL}_{\text{pk},\delta_{\text{max}}-1}) \in \mathbb{G}_q^{\delta_{\text{max}}}$~~ Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\text{sup}}]$ \triangleright Can be derived from \mathbf{pTable} using algorithm 3.9
~~Choice Return Codes encryption public key $\mathbf{pk}_{\text{CCR}} \in \mathbb{G}_q^{\psi_{\text{max}}}$~~ Extracted election event $\mathbf{eee} = (\mathbf{hContext}, (p, q, g), \mathbf{ee}, \mathbf{evcs})$

Stateful Lists and Maps:

List of voting cards with decrypted partial Choice Return Codes $L_{\text{decPCC},j}$

Input:

Encrypted vote $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$
 Exponentiated encrypted vote $\widetilde{\mathbf{E1}} = (\gamma_1^{\mathbf{k}_{\text{id}}}, \phi_{1,0}^{\mathbf{k}_{\text{id}}}) \in \mathbb{G}_q^2$
 Encrypted partial Choice Return Codes $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$
 CCR _{j} Choice Return Codes encryption secret key $(\mathbf{sk}_{\text{CCR},j,0}, \dots, \mathbf{sk}_{\text{CCR},j,\psi_{\text{max}}-1}) \in \mathbb{Z}_q^{\psi_{\text{max}}}$
 CCR _{j} Choice Return Codes encryption public key $(\mathbf{pk}_{\text{CCR},j,0}, \dots, \mathbf{pk}_{\text{CCR},j,\psi_{\text{max}}-1}) \in \mathbb{G}_q^{\psi_{\text{max}}}$

Require: $\mathbf{vc}_{\text{id}} \notin L_{\text{decPCC},j}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: ~~$\mathbf{i}_{\text{aux}} \leftarrow (\text{"PartialDecryptPCC"}, \mathbf{vc}_{\text{id}}, \text{GetHashContext}())$~~ $\mathbf{i}_{\text{aux}} \leftarrow (\text{"PartialDecryptPCC"}, \mathbf{vc}_{\text{id}}, \text{GetHashContext}())$ \triangleright See algorithm 3.15
 - 2: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$
 - 3: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1^{\mathbf{k}_{\text{id}}}), \text{IntegerToString}(\phi_{1,0}^{\mathbf{k}_{\text{id}}}))$
 - 4: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}), \dots, \text{IntegerToString}(\phi_{2,\psi-1}))$
 - 5: $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(j))$
 - 6: **for** $i \in [0, \psi)$ **do**
 - 7: $d_{j,i} \leftarrow \gamma_2^{\mathbf{sk}_{\text{CCR},j,i}} \bmod p$
 - 8: $\pi_{\text{decPCC},j,i} \leftarrow \text{GenExponentiationProof}((g, \gamma_2), \mathbf{sk}_{\text{CCR},j,i}, (\mathbf{pk}_{\text{CCR},j,i}, d_{j,i}), \mathbf{i}_{\text{aux}})$
 - 9: **end for**
 - 10: ~~$L_{\text{decPCC},j} \leftarrow L_{\text{decPCC},j} \cup \mathbf{vc}_{\text{id}}$~~ $L_{\text{decPCC},j} \leftarrow L_{\text{decPCC},j} \parallel \mathbf{vc}_{\text{id}}$
-

Output:

Exponentiated γ elements $\mathbf{d}_j = (d_{j,0}, \dots, d_{j,\psi-1}) \in \mathbb{G}_q^{\psi}$
 Exponentiation proofs $\boldsymbol{\pi}_{\text{decPCC},j} = (\pi_{\text{decPCC},j,0}, \dots, \pi_{\text{decPCC},j,\psi-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi}$

5.2.4 DecryptPCC

Algorithm 5.7 DecryptPCC

Context:

~~Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$~~ The CCR's index $j \in [1, 4]$
The other CCR's indices $\hat{\mathbf{j}} = (\hat{j}_1, \hat{j}_2, \hat{j}_3) = (1, 2, 3, 4) \setminus j$
~~Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{in}}$ Verification card set ID $\mathbf{ves} \in (\mathbb{A}_{Base16})^{1_{in}}$~~ Verification card ID $\mathbf{vc}_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$
~~Number of allowed selections for this specific ballot box $\psi \in [1, \psi_{sup}]$~~ \triangleright Can be derived from pTable using algorithm 3.8
~~Election public key $\mathbf{EL}_{pk} = (\mathbf{EL}_{pk,0}, \dots, \mathbf{EL}_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$~~ ~~Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{sup}]$~~ \triangleright Can be derived from pTable using algorithm 3.9
~~Choice Return Codes encryption public key $\mathbf{pk}_{CCR} \in \mathbb{G}_q^{\psi_{max}}$~~ Extracted election event $\mathbf{eee} = (\mathbf{hContext}, (p, q, g), \mathbf{ee}, \mathbf{evcs})$
Other CCR's Choice Return Codes encryption keys $(\mathbf{pk}_{CCR_{\hat{j}_1}}, \mathbf{pk}_{CCR_{\hat{j}_2}}, \mathbf{pk}_{CCR_{\hat{j}_3}}) \in (\mathbb{G}_q^{\psi_{max}})^3$

Input:

CCR $_j$'s exponentiated γ elements $\mathbf{d}_j = (d_{j,0}, \dots, d_{j,\psi-1}) \in \mathbb{G}_q^\psi$
Other CCR's exponentiated γ elements $(\mathbf{d}_{\hat{j}_1}, \mathbf{d}_{\hat{j}_2}, \mathbf{d}_{\hat{j}_3}) \in (\mathbb{G}_q^\psi)^3$
Other CCR's exponentiation proofs $(\pi_{decPCC,\hat{j}_1}, \pi_{decPCC,\hat{j}_2}, \pi_{decPCC,\hat{j}_3}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi \times 3}$
Encrypted vote $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$
Exponentiated encrypted vote $\mathbf{\bar{E}1} = (\gamma_1^{k_{id}}, \phi_{1,0}^{k_{id}}) \in \mathbb{G}_q^2$
Encrypted partial Choice Return Codes $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: for  $k \in \hat{\mathbf{j}}$  do
2:    $\mathbf{i}_{aux} \leftarrow (\text{"PartialDecryptPCC"}, \mathbf{vc}_{id}, \text{GetHashContext}())$   $\mathbf{i}_{aux} \leftarrow (\text{"PartialDecryptPCC"}, \mathbf{vc}_{id}, \text{GetHashExtractedElectionEvent}())$ 
 $\triangleright$  See algorithm 3.15
3:    $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$ 
4:    $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(\gamma_1^{k_{id}}), \text{IntegerToString}(\phi_{1,0}^{k_{id}}))$ 
5:    $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}), \dots, \text{IntegerToString}(\phi_{2,\psi-1}))$ 
6:    $\mathbf{i}_{aux} \leftarrow (\mathbf{i}_{aux}, \text{IntegerToString}(k))$ 
7:    $\mathbf{pk}_{CCR_k} = (\mathbf{pk}_{CCR_{k,0}}, \dots, \mathbf{pk}_{CCR_{k,\psi_{max}-1}})$ 
8:    $\mathbf{d}_k = (d_{k,0}, \dots, d_{k,\psi-1})$ 
9:    $\pi_{decPCC,k} = (\pi_{decPCC,k,0}, \dots, \pi_{decPCC,k,\psi-1})$ 
10:  for  $i \in [0, \psi]$  do
11:    if  $\neg \text{VerifyExponentiation}((g, \gamma_2), (\mathbf{pk}_{CCR_{k,i}}, d_{k,i}), \pi_{decPCC,k,i}, \mathbf{i}_{aux})$  then
12:      return  $\perp$ 
13:    end if
14:  end for
15: end for
16: for  $i \in [0, \psi]$  do
17:    $d_i \leftarrow (d_{j,i} \cdot d_{\hat{j}_1,i} \cdot d_{\hat{j}_2,i} \cdot d_{\hat{j}_3,i}) \bmod p$ 
18:    $\mathbf{pCC}_{id,i} \leftarrow \frac{\phi_{2,i}}{d_i} \bmod p$ 
19: end for
20:  $\mathbf{pCC}_{id} \leftarrow (\mathbf{pCC}_{id,0}, \dots, \mathbf{pCC}_{id,\psi-1})$ 

```

Output:

```

21: if all control components' zero-knowledge proofs verify then
22:    $\mathbf{pCC}_{id} = (\mathbf{pCC}_{id,0}, \dots, \mathbf{pCC}_{id,\psi-1}) \in \mathbb{G}_q^\psi$ 
23: else
24:    $\perp$ , the verification of at least one  $\pi_{decPCC,j,i}$  failed.
25: end if

```

5.2.5 CreateLCCShare

Upon receipt of the other control components' contributions \mathbf{d}_j and $\pi_{\text{decPCC},j}$, the Return Codes control components CCR check the zero-knowledge proofs, combine the inputs, and generate long Choice Return Codes shares using the voting client's partial Choice Return Codes \mathbf{pCC}_{id} and the CCR_j Return Codes Generation secret key \mathbf{k}'_j . Each of the four control components $j \in [1, 4]$ executes this algorithm. The control components use the following lists:

- list of voting cards $\mathbf{L}_{\text{decPCC},j}$ for which the control components decrypted the partial Choice Return Codes \mathbf{pCC}_{id} ,
- list of voting cards $\mathbf{L}_{\text{sentVotes},j}$ for which the control components already generated long Choice Return Code shares,
- the partial Choice Return Codes allow list \mathbf{L}_{pcc} .

Importantly, the **CreateLCCShare** returns *no* CCR_j 's long Choice Return Code shares $\mathbf{lCC}_{j,\text{id},i}$ if one or more validations within the algorithm fail.

Moreover, the honest control component guarantees the following before generating the long Choice Return Code shares $\mathbf{lCC}_{j,\text{id}}$:

- the zero-knowledge proofs ascertain that the product of partial Choice Return Codes correspond to the exponentiated encrypted vote (see algorithm **VerifyBallotCCR**),
- the other control components correctly decrypted the partial Choice Return Codes (see algorithm **DecryptPCC**),
- every partial Choice Return Code is valid since each corresponds to an entry in the partial Choice Return Codes allow list \mathbf{L}_{pcc} ,
- the combination of partial Choice Return Codes is correct.

These verifications prevent attacks against the malleability of the zero-knowledge proofs (see Haines et al. [14]) and ensure that the honest control components detect attacks against individual verifiability at the time of voting (see [Gitlab issue 7](#)).

Algorithm 5.8 CreateLCCShare

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 The CCR's index $j \in [1, 4]$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card ID $\mathbf{vc}_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Blank correctness information $\hat{\tau} = (\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1}) \in (\mathcal{T}_1^{50})^\psi$ \triangleright Can be derived from \mathbf{pTable} using algorithm 3.6

Stateful Lists and Maps:

List of voting cards with decrypted partial Choice Return Codes $\mathbf{L}_{decPCC,j}$

CCR_j's list of sent voting cards $\mathbf{L}_{sentVotes,j}$

Input:

Partial Choice Return Codes allow list $\mathbf{L}_{pcc} \in (\mathbb{A}_{Base64}^{1_{HB64}})^{n \cdot N_E}$

A vector of partial Choice Return Codes $\mathbf{pCC}_{id} \in (\mathbb{G}_q)^\psi$

CCR_j Return Codes Generation secret key $\mathbf{k}'_j \in \mathbb{Z}_q$

Require: $\mathbf{pCC}_{id,i} \neq \mathbf{pCC}_{id,k}, \forall i, k \in \{0, \dots, (\psi - 1)\} \wedge i \neq k$ \triangleright All pCC must be distinct

Require: $\mathbf{vc}_{id} \in \mathbf{L}_{decPCC,j}$

Require: $\mathbf{vc}_{id} \notin \mathbf{L}_{sentVotes,j}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1:  $\mathbf{PRK} \leftarrow \text{IntegerToByteArray}(\mathbf{k}'_j)$ 
2:  $\mathbf{info} \leftarrow (\text{"VoterChoiceReturnCodeGeneration"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{id})$ 
3:  $\mathbf{k}_{j,id} \leftarrow \text{KDFToZq}(\mathbf{PRK}, \mathbf{info}, q)$ 
4: for  $i \in [0, \psi)$  do
5:    $\mathbf{hpCC}_{id,i} \leftarrow \text{HashAndSquare}(\mathbf{pCC}_{id,i})$ 
6:    $\mathbf{lpCC}_{id,i} \leftarrow \text{RecursiveHash}(\mathbf{hpCC}_{id,i}, \mathbf{vc}_{id}, \mathbf{ee}, \hat{\tau}_i)$ 
7:   if  $\text{Base64Encode}(\mathbf{lpCC}_{id,i}) \notin \mathbf{L}_{pcc}$  then
8:     return  $\perp$ 
9:   else
10:     $\mathbf{lCC}_{j,id,i} \leftarrow \mathbf{hpCC}_{id,i}^{\mathbf{k}_{j,id}} \bmod p$ 
11:  end if
12: end for
13:  $\mathbf{lCC}_{j,id} \leftarrow (\mathbf{lCC}_{j,id,0}, \dots, \mathbf{lCC}_{j,id,\psi-1})$ 
14:  $\mathbf{L}_{sentVotes,j} \leftarrow \mathbf{L}_{sentVotes,j} \cup \mathbf{vc}_{id}$   $\mathbf{L}_{sentVotes,j} \leftarrow \mathbf{L}_{sentVotes,j} || \mathbf{vc}_{id}$ 

```

Output:

CCR_j's long Choice Return Code share $\mathbf{lCC}_{j,id} = (\mathbf{lCC}_{j,id,0}, \dots, \mathbf{lCC}_{j,id,\psi-1}) \in \mathbb{G}_q^\psi$

5.2.6 ExtractCRC

The voting server extracts the short Choice Return Codes \mathbf{CC}_{id} from the Return Codes Mapping table $\mathbf{CMtable}$. Subsequently, the voting server sends the short Choice Return Codes \mathbf{CC}_{id} to the voting client and the voter compares them to the one printed on the voting card.

Algorithm 5.9 ExtractCRC

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}}$
 Verification card ID $\mathbf{vc}_{\text{id}} \in (\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}}$
 Blank correctness information $\hat{\tau} = (\hat{\tau}_0, \dots, \hat{\tau}_{\psi-1}) \in (\mathcal{T}_1^{50})^\psi \triangleright$ Can be derived from \mathbf{pTable} using algorithm 3.6

Input:

CCR long Choice Return Code shares $(\mathbf{lCC}_{1,\text{id}}, \mathbf{lCC}_{2,\text{id}}, \mathbf{lCC}_{3,\text{id}}, \mathbf{lCC}_{4,\text{id}}) \in (\mathbb{G}_q^\psi)^4$
 Return Codes Mapping table $\mathbf{CMtable} \in (\mathbb{A}_{\text{Base64}}^{1_{\text{HB64}}} \times \mathbb{A}_{\text{Base64}}^*)^{N_{\text{E}} \cdot (n+1)}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: for  $i \in [0, \psi)$  do
2:    $\mathbf{pC}_{\text{id},i} \leftarrow \prod_{j=1}^4 \mathbf{lCC}_{j,\text{id},i} \bmod p$ 
3:    $\mathbf{lCC}_{\text{id},i} \leftarrow \text{RecursiveHash}(\mathbf{pC}_{\text{id},i}, \mathbf{vc}_{\text{id}}, \mathbf{ee}, \hat{\tau}_i)$ 
4:    $\mathbf{key} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(\mathbf{lCC}_{\text{id},i}))$ 
5:   if  $\mathbf{key} \notin \mathbf{CMtable}$  then
6:     return  $\perp$ 
7:   else
8:      $\mathbf{ctCC}_{\text{id},i,\text{encoded}} \leftarrow \mathbf{CMtable}(\mathbf{key}) \quad \triangleright$  Retrieve the value from the key-value map
9:      $\mathbf{ctCC}_{\text{id},i,\text{combined}} \leftarrow \text{Base64Decode}(\mathbf{ctCC}_{\text{id},i,\text{encoded}})$ 
10:     $\mathbf{length} \leftarrow$  Length in bytes of  $\mathbf{ctCC}_{\text{id},i,\text{combined}}$ 
11:     $\mathbf{split} \leftarrow \mathbf{length} - \mathbf{nonceLength} \quad \triangleright$  Nonce length in bytes defined by the
    symmetric algorithm used; see crypto primitives specification
12:     $\mathbf{ctCC}_{\text{id},i,\text{ciphertext}} \leftarrow \mathbf{ctCC}_{\text{id},i,\text{combined}}[0 : \mathbf{split}]$ 
13:     $\mathbf{ctCC}_{\text{id},i,\text{nonce}} \leftarrow \mathbf{ctCC}_{\text{id},i,\text{combined}}[\mathbf{split} : \mathbf{length}]$ 
14:     $\mathbf{skcc}_{\text{id},i} \leftarrow \text{KDF}(\mathbf{lCC}_{\text{id},i}, (), \mathbf{l}_{\text{KD}})$ 
15:     $\mathbf{CC}_{\text{id},i,\text{bytes}} \leftarrow \text{GetPlaintextSymmetric}(\mathbf{skcc}_{\text{id},i}, \mathbf{ctCC}_{\text{id},i,\text{ciphertext}}, \mathbf{ctCC}_{\text{id},i,\text{nonce}}, ())$ 
16:     $\mathbf{CC}_{\text{id},i} \leftarrow \text{ByteArrayToString}(\mathbf{CC}_{\text{id},i,\text{bytes}})$ 
17:  end if
18: end for
19:  $\mathbf{CC}_{\text{id}} \leftarrow (\mathbf{CC}_{\text{id},0}, \dots, \mathbf{CC}_{\text{id},\psi-1})$ 

```

Output:

Short Choice Return Codes $\mathbf{CC}_{\text{id}} \in ((\mathbb{A}_{10})^{1_{\text{cc}}})^\psi \triangleright$ The algorithm returns the short Choice Return Codes only if *all* of them are extractable.

5.3 ~~ConfirmVote~~ConfirmVote

After the voter successfully checked the short Choice Return Codes, the ConfirmVote sub-protocol comprises confirming the vote until receiving the Vote Cast Return Code. Figure 12 highlights the algorithms and data flows.

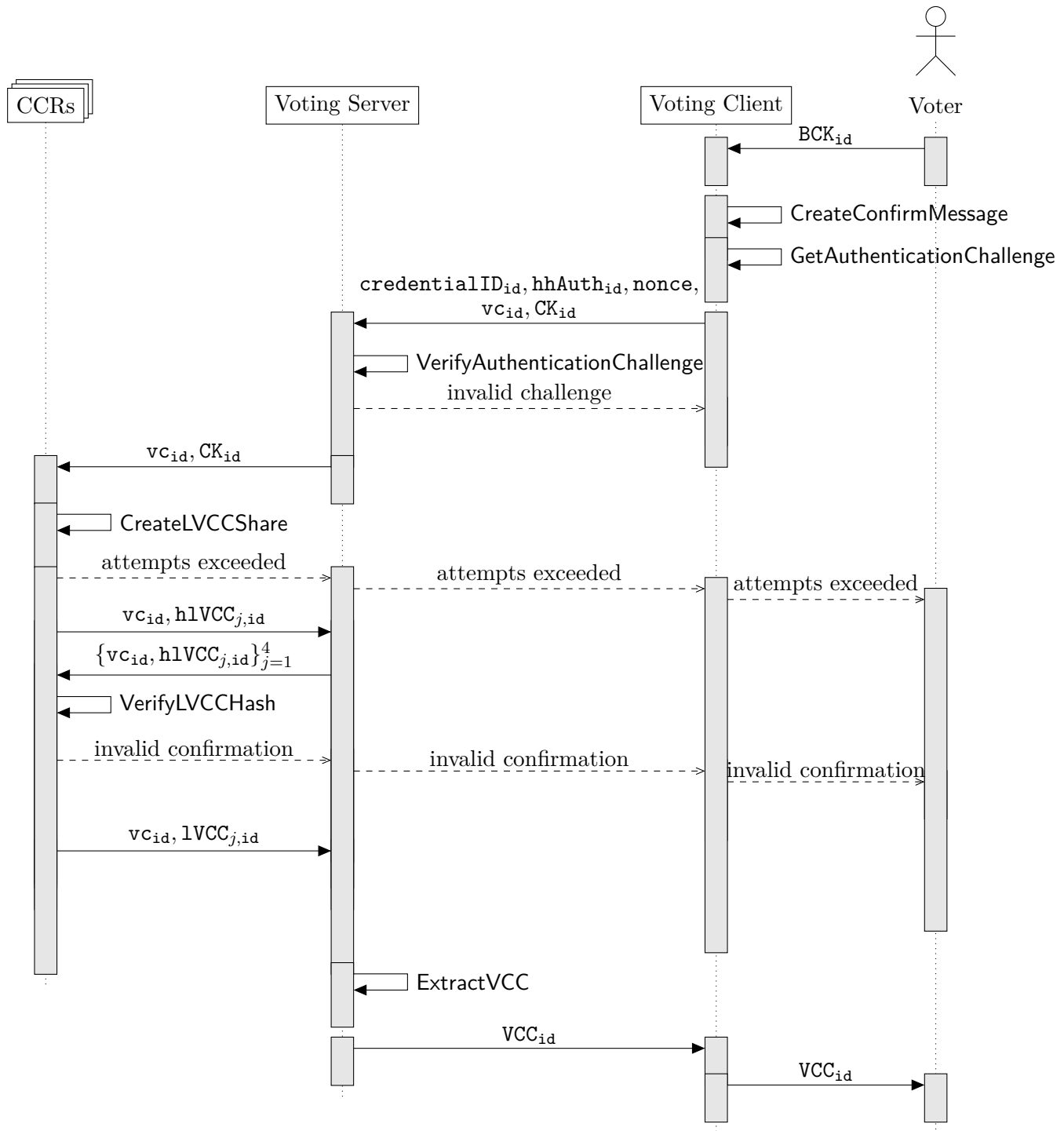


Fig. 12: Sequence diagram of the ConfirmVote sub-protocol.

5.3.1 CreateConfirmMessage

The voter enters the Ballot Casting Key BCK_{id} in the voting client, which executes the CreateConfirmMessage algorithm.

Algorithm 5.10 CreateConfirmMessage

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

Ballot Casting Key $BCK_{id} \in (\mathbb{A}_{10})^{1_{BCK}}$ \triangleright Must contain one non-zero element
Verification Card Secret Key $k_{id} \in \mathbb{Z}_q$

Operation:

1: $hBCK_{id} \leftarrow \text{HashAndSquare}(\text{StringToInteger}(BCK_{id}))$ \triangleright See crypto primitives specification
2: $CK_{id} \leftarrow hBCK_{id}^{k_{id}} \bmod p$

Output:

Confirmation Key $CK_{id} \in \mathbb{G}_q$

5.3.2 CreateLVCCShare

The control components execute the **CreateLVCCShare** algorithm. **CreateLVCCShare** is similar to **CreateLCCShare** (see section 5.2.5). However, while the control components run **CreateLCCShare** at most once, the control components allow up to five executions of **CreateLVCCShare** — in case the voter entered an invalid ballot casting key. Furthermore, the control components derive their share using the Voter Vote Cast Return Code Generation secret key $\mathbf{kc}_{j,\text{id}}$ instead of the Voter Choice Return Code Generation secret key $\mathbf{k}_{j,\text{id}}$ to prevent the adversary from using the **CreateLVCCShare** algorithm as an oracle to learn Choice Return Codes.

Algorithm 5.11 CreateLVCCShare

Context:

- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$
- The CCR's index $j \in [1, 4]$
- Election event ID $\mathbf{ee} \in (\mathbb{A}_{\text{Base16}})^{1\text{ID}}$
- Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{\text{Base16}})^{1\text{ID}}$
- Verification card ID $\mathbf{vc}_{\text{id}} \in (\mathbb{A}_{\text{Base16}})^{1\text{ID}}$

Stateful Lists and Maps:

- CCR_j's list of sent voting cards $\mathbf{L}_{\text{sentVotes},j}$
- CCR_j's key-value map of number of confirmation attempts per verification card $\mathbf{L}_{\text{confirmationAttempts},j}$
- CCR_j's list of confirmed voting cards $\mathbf{L}_{\text{confirmedVotes},j}$

Input:

- Confirmation Key $\mathbf{CK}_{\text{id}} \in \mathbb{G}_q$
- CCR_j Return Codes Generation secret key $\mathbf{k}'_j \in \mathbb{Z}_q$

Require: $\mathbf{vc}_{\text{id}} \in \mathbf{L}_{\text{sentVotes},j}$

Require: $\mathbf{vc}_{\text{id}} \notin \mathbf{L}_{\text{confirmedVotes},j}$

Operation:

- 1: $\mathbf{attempts}_{\text{id}} \leftarrow \mathbf{L}_{\text{confirmationAttempts},j}(\mathbf{vc}_{\text{id}})$ ▷ Retrieve the value from the key-value map
 - 2: **if** $\mathbf{attempts}_{\text{id}} \geq 5$ **then return** \perp ▷ The voter has only 5 attempts to confirm the vote.
 - 3: **end if**
 - 4: $\mathbf{PRK} \leftarrow \text{IntegerToByteArray}(\mathbf{k}'_j)$
 - 5: $\mathbf{info}_{\mathbf{CK}} \leftarrow (\text{"VoterVoteCastReturnCodeGeneration"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{\text{id}})$
 - 6: $\mathbf{kc}_{j,\text{id}} \leftarrow \text{KDFToZq}(\mathbf{PRK}, \mathbf{info}_{\mathbf{CK}}, q)$
 - 7: $\mathbf{hCK}_{\text{id}} \leftarrow \text{HashAndSquare}(\mathbf{CK}_{\text{id}})$
 - 8: $\mathbf{1VCC}_{j,\text{id}} \leftarrow \mathbf{hCK}_{\text{id}}^{\mathbf{kc}_{j,\text{id}}} \bmod p$
 - 9: $\mathbf{i}_{\text{aux}} \leftarrow (\text{"CreateLVCCShare"}, \mathbf{ee}, \mathbf{vcs}, \mathbf{vc}_{\text{id}}, \text{IntegerToString}(j))$
 - 10: $\mathbf{h1VCC}_{j,\text{id}} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(\mathbf{i}_{\text{aux}}, \mathbf{1VCC}_{j,\text{id}}))$
 - 11: $\mathbf{L}_{\text{confirmationAttempts},j}(\mathbf{vc}_{\text{id}}) \leftarrow \mathbf{attempts}_{\text{id}} + 1$ ▷ Set the value in the key-value map
-

Output:

- CCR_j's long Vote Cast Return Code share $\mathbf{1VCC}_{j,\text{id}} \in \mathbb{G}_q$
 - CCR_j's hashed long Vote Cast Return Code share $\mathbf{h1VCC}_{j,\text{id}} \in \mathbb{A}_{\text{Base64}}^{1\text{HB64}}$
 - Confirmation attempt number $\mathbf{attempts}_{\text{id}} \in [0, 4]$
-

5.3.3 VerifyLVCCHash

The VerifyLVCCHash algorithm allows the control components to determine if the confirmation attempt was successful—or if the confirmation attempt corresponds to an incorrect Ballot Casting Key BCK_{id} .

Only if the CCR_j 's hashed long Vote Cast Return Code share $hlVCC_{j,id}$ is extractable from the long Vote Cast Return Codes allow list L_{lvcc} , do the control components return their CCR_j 's long Vote Cast Return Code share $lvCC_{j,id}$ to the voting server.

Algorithm 5.12 VerifyLVCCHash

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 The CCR's index $j \in [1, 4]$
 The other CCR's indices $\hat{j} = (\hat{j}_1, \hat{j}_2, \hat{j}_3) = (1, 2, 3, 4) \setminus j$
 Election event ID $ee \in (\mathbb{A}_{Base16})^{1ID}$
 Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1ID}$
 Verification card ID $vc_{id} \in (\mathbb{A}_{Base16})^{1ID}$

Stateful Lists and Maps:

CCR_j 's list of sent voting cards $L_{sentVotes,j}$
 CCR_j 's list of confirmed voting cards $L_{confirmedVotes,j}$

Input:

Long Vote Cast Return Codes allow list $L_{lvcc} \in (\mathbb{A}_{Base64}^{1_{HB64}})^{N_E}$
 CCR_j 's hashed long Vote Cast Return Code share $hlVCC_{j,id} \in \mathbb{A}_{Base64}^{1_{HB64}}$
 Other CCR's hashed long Vote Cast Return Code shares $(hlVCC_{\hat{j}_1,id}, hlVCC_{\hat{j}_2,id}, hlVCC_{\hat{j}_3,id}) \in (\mathbb{A}_{Base64}^{1_{HB64}})^3$

Require: $vc_{id} \in L_{sentVotes,j}$

Require: $vc_{id} \notin L_{confirmedVotes,j}$

Operation:

▷ For all algorithms see the crypto primitives specification

```

1:  $i_{aux} \leftarrow (\text{"VerifyLVCCHash"}, ee, vcs, vc_{id})$ 
2:  $hhVCC_{id} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(i_{aux}, hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}))$ 
3: if  $hhVCC_{id} \in L_{lvcc}$  then
4:    $L_{confirmedVotes,j} \leftarrow L_{confirmedVotes,j} \cup (vc_{id})$   $L_{confirmedVotes,j} \leftarrow L_{confirmedVotes,j} || vc_{id}$ 
5:   return  $\top$ 
6: else
7:   return  $\perp$ 
8: end if
```

Output:

The result of the verification: \top if the verification is successful and an updated list of $L_{confirmedVotes,j}$, \perp otherwise.

The control component sends the CCR_j 's long Vote Cast Return Code share $lvCC_{j,id}$ to the voting server only upon a successful confirmation attempt. Otherwise, the control component returns a message that the confirmation attempt was not successful.

5.3.4 ExtractVCC

In case that the confirmation attempt was successful, the voting server extracts the short Vote Cast Return Code VCC_{id} from the Return Codes Mapping table $CMtable$.

Algorithm 5.13 ExtractVCC

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card ID $vc_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Input:

CCR long Vote Cast Return Code shares $(1VCC_{1,id}, 1VCC_{2,id}, 1VCC_{3,id}, 1VCC_{4,id}) \in \mathbb{G}_q^4$
 Return Codes Mapping table $CMtable \in (\mathbb{A}_{Base64}^{1_{HB64}} \times \mathbb{A}_{Base64}^*)^{N_E \cdot (n+1)}$

Operation:

▷ For all algorithms see the crypto primitives specification

```

1:  $pVCC_{id} \leftarrow \prod_{j=1}^4 1VCC_{j,id} \bmod p$ 
2:  $1VCC_{id} \leftarrow \text{RecursiveHash}(pVCC_{id}, vc_{id}, ee)$ 
3:  $key \leftarrow \text{Base64Encode}(\text{RecursiveHash}(1VCC_{id}))$ 
4: if  $key \notin CMtable$  then
5:   return  $\perp$ 
6: else
7:    $ctVCC_{id,encoded} \leftarrow CMtable(key)$  ▷ Retrieve the value from the key-value map
8:    $ctVCC_{id,combined} \leftarrow \text{Base64Decode}(ctVCC_{id,encoded})$ 
9:    $length \leftarrow \text{Length in bytes of } ctVCC_{id,combined}$ 
10:   $split \leftarrow length - nonceLength$  ▷ Nonce length in bytes defined by the symmetric
    algorithm used
11:   $ctVCC_{id,ciphertext} \leftarrow ctVCC_{id,combined}[0 : split]$ 
12:   $ctVCC_{id,nonce} \leftarrow ctVCC_{id,combined}[split : length]$ 
13:   $skvcc_{id} \leftarrow \text{KDF}(1VCC_{id}, (), 1_{KD})$ 
14:   $VCC_{id,bytes} \leftarrow \text{GetPlaintextSymmetric}(skvcc_{id}, ctVCC_{id,ciphertext}, ctVCC_{id,nonce}, ())$ 
15:   $VCC_{id} \leftarrow \text{ByteArrayToString}(VCC_{id,bytes})$ 
16: end if

```

Output:

Short Vote Cast Return Code $VCC_{id} \in (\mathbb{A}_{10})^{1_{VCC}}$

Subsequently, the voting server sends the short Vote Cast Return Code VCC_{id} to the voting client and the voter compares it to the one printed on the voting card.

As soon as the voting server successfully performs the **ExtractVCC** algorithm, the voting card status changes, and the voter can no longer vote by post or in person. Conversely, if the voting server did *not* execute the **ExtractVCC** algorithm, the voter can still vote by post or in person. The Federal Chancellery's Ordinance [7] and its explanatory report [8] elaborate on that requirement.

[VEleS Annex 4.11]: As long as the system has not registered confirmation of a definitive electronic vote, the voter may still choose to cast his or her vote via a conventional voting channel.

[Explanatory Report, Sec 4.2.1]: No 4.11: Voters are required to report to the competent cantonal authority if proofs are incorrectly displayed or if they are unsure about this. Voting by post or in person remains an option if an electronic vote has not yet been received. In order to assess this, the cantons have functionality at their disposal in accordance with Number 11.6.

The voting server allows the competent cantonal authority to query a voter's voting card status.

For this specific functionality, the e-voting system can be considered *trustworthy* as highlighted in the the Federal Chancellery's Ordinance [7] and its explanatory report [8].

[VEleS Annex 11.6]: The system allows the polling card to be used to determine whether someone has cast an electronic vote.

[Explanatory Report, Sec 4.2.1]: No 11.6: It is not possible to decide whether a vote cast by post or in person is a double or even multiple vote by using only the votes cast electronically as a basis for comparison. Nevertheless, the functionality under Number 11.6 falls within the scope of the OEV. However, it is not necessary to specify the functionality by reference to trust assumptions under Number 2.

When a ballot is cast by post or in person (paper vote), the cantonal or municipal authorities check the status of the associated voting card. The cantonal or municipal authorities will refuse the paper vote if the voter has already cast an electronic vote. If not, the paper vote will be cast, and the electronic channel will be blocked through a change of the voting card status.

6 Tally Phase

The tally phase’s purpose is for each control component to sequentially shuffle and decrypt the encrypted votes—in a verifiable manner—allowing an auditor using a trustworthy verifier software to check that the election result is *verifiably* correct.

We distinguish between the operations in the online control components (**MixOnline**) and the offline Tally control component (**MixOffline**). Table 12 shows the algorithms involved.

Algorithm	Actor	Reference
MixOnline		Figures 13 and 14
GetMixnetInitialCiphertexts	Online Control Component (CCM)	6.1
VerifyMixDecOnline	Online Control Component (CCM)	6.2
MixDecOnline	Online Control Component (CCM)	6.3
MixOffline		Figure 13
VerifyVotingClientProofs	Tally Control Component	6.9
VerifyMixDecOffline	Tally Control Component	6.10
MixDecOffline	Tally Control Component	6.11
ProcessPlaintexts	Tally Control Component	6.12
VerifyTally	Auditors	Verifier Specification [21]

Tab. 14: Overview of the algorithms in the tally phase.

Conceptually, it is important to note that each control component generates a CCM election key pair.

$$(\text{EL}_{\text{pk},j}, \text{EL}_{\text{sk},j}) \leftarrow \text{GenKeyPair}$$

$$\text{EL}_{\text{pk},j} = g^{\text{EL}_{\text{sk},j}} \bmod p$$

The election public key is the product of all CCM election public keys and the electoral board public key EB_{pk} .

$$\text{EL}_{\text{pk}} = \prod_{j=1}^4 \text{EL}_{\text{pk},j} \cdot \text{EB}_{\text{pk}} \bmod p = g^{\text{EL}_{\text{sk},1} + \text{EL}_{\text{sk},2} + \text{EL}_{\text{sk},3} + \text{EL}_{\text{sk},4} + \text{EB}_{\text{sk}}} \bmod p$$

Now, imagine that the first control component partially decrypted the ciphertexts. The second control component needs to shuffle the ciphertexts using the *remaining* election public key $\overline{\text{EL}}_{\text{pk}}$. For the second control component, the remaining public key does not contain the first control component’s contribution:

$$\overline{\text{EL}}_{\text{pk},2} = \text{EL}_{\text{pk},2} \cdot \text{EL}_{\text{pk},3} \cdot \text{EL}_{\text{pk},4} \cdot \text{EB}_{\text{pk}} \bmod p = \frac{\text{EL}_{\text{pk}}}{\text{EL}_{\text{pk},1}} \bmod p = g^{\text{EL}_{\text{sk},2} + \text{EL}_{\text{sk},3} + \text{EL}_{\text{sk},4} + \text{EB}_{\text{sk}}} \bmod p$$

The same principle holds for the subsequent shuffle and decryption operations. Figure 13 indicates the data flows and algorithms in the tally phase.

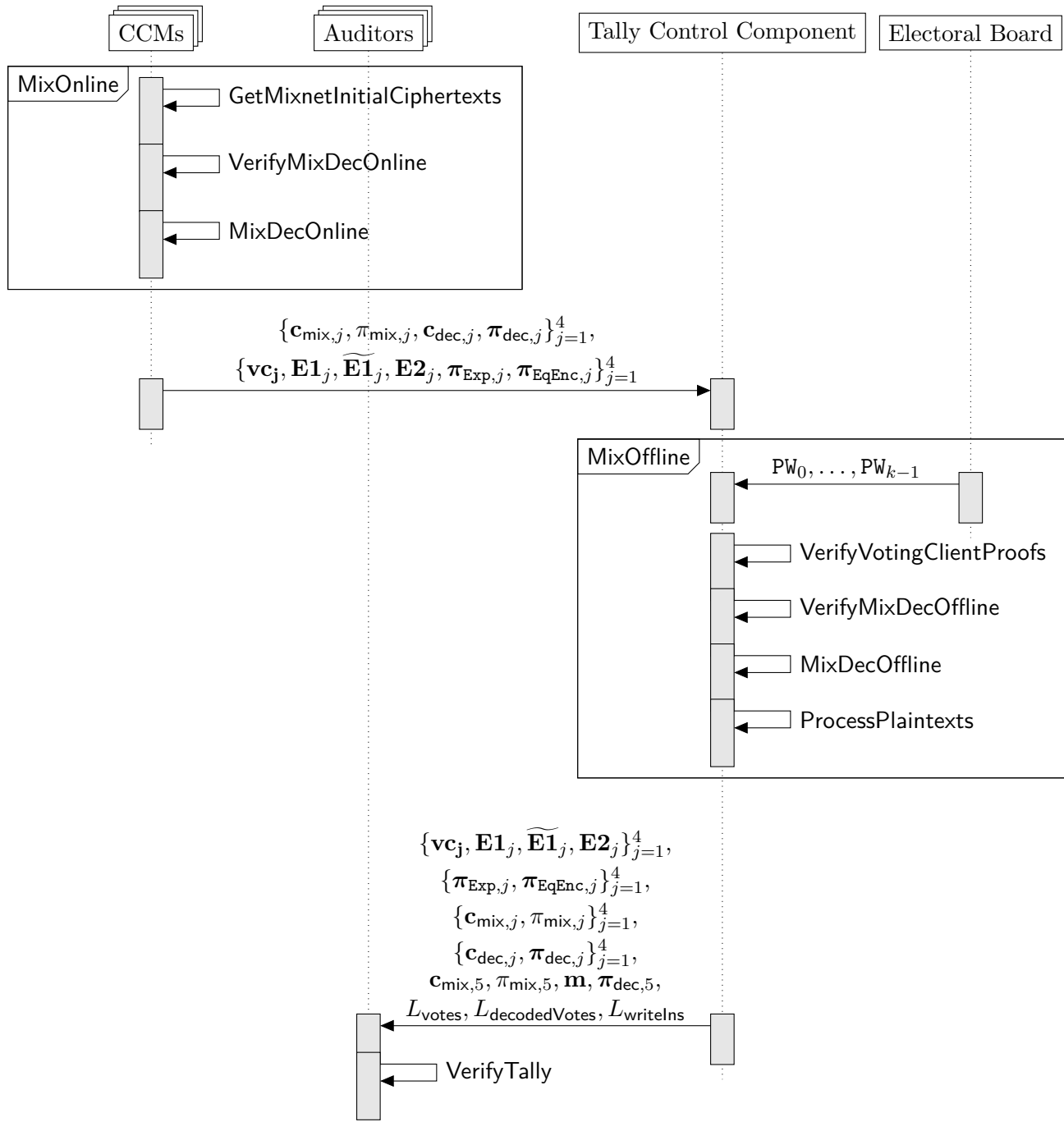


Fig. 13: Sequence diagram of the tally phase showing the data flows and involved algorithms.

6.1 MixOnline

The online control components execute the first part of the tally phase. Figure 14 highlights the algorithms and data flows. The CCM_1 does not execute the ~~VerifyMixDecOnline~~ VerifyMixDecOnline algorithm, since the first mixing control component has nothing to verify. Each online control component sends back the encrypted confirmed votes registered during the voting phase for later processing by the Tally Control Component and auditors.

The j -th control component's encrypted confirmed votes contains the following data:

- j -th control component's list of confirmed verification card IDs $\mathbf{vc}_j = (\mathbf{vc}_{j,0}, \dots, \mathbf{vc}_{j,N_C-1})$
- j -th control component's list of encrypted, confirmed votes $\mathbf{E1}_j = (\mathbf{E1}_{j,0}, \dots, \mathbf{E1}_{j,N_C-1})$
- j -th control component's list of exponentiated, encrypted, confirmed votes $\widetilde{\mathbf{E1}}_j = (\widetilde{\mathbf{E1}}_{j,0}, \dots, \widetilde{\mathbf{E1}}_{j,N_C-1})$
- j -th control component's list of encrypted, partial Choice Return Codes $\mathbf{E2}_j = (\mathbf{E2}_{j,0}, \dots, \mathbf{E2}_{j,N_C-1})$
- j -th control component's list of exponentiation proofs $\boldsymbol{\pi}_{\text{Exp},j} = (\pi_{\text{Exp},j,0}, \dots, \pi_{\text{Exp},j,N_C-1})$
- j -th control component's list of plaintext equality proofs $\boldsymbol{\pi}_{\text{EqEnc},j} = (\pi_{\text{EqEnc},j,0}, \dots, \pi_{\text{EqEnc},j,N_C-1})$

Moreover, the control components ensure that the ballot box cannot be decrypted before the election event period ends, except for test ballot boxes which can be decrypted at any time (see section 3.4).

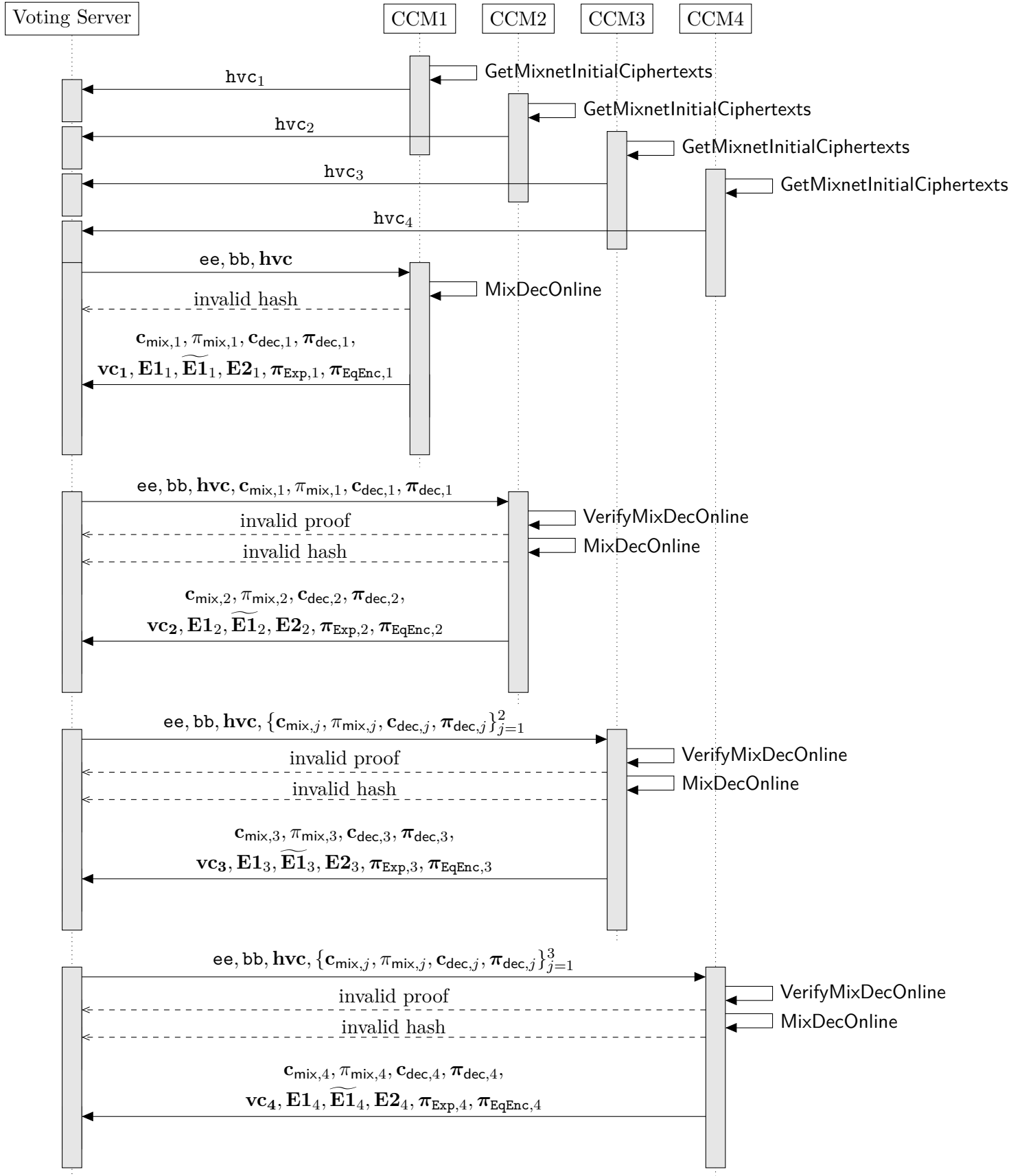


Fig. 14: Sequence diagram of the MixOnline protocol.

6.1.1 GetMixnetInitialCiphertexts

The control components retrieve the mixnet's initial ciphertexts from the list of confirmed votes $L_{\text{confirmedVotes},j}$ that they established during the voting phase. To this end, the control components retrieve a key-value map linking the verification card IDs to the encrypted, confirmed votes $\mathbf{vcMap}_j = ((\mathbf{vc}_1, \mathbf{E1}_{j,1}) \dots, (\mathbf{vc}_{N_C-1}, \mathbf{E1}_{j,N_C-1}))$ and provide this map to the algorithm **GetMixnetInitialCiphertexts**. Moreover, shuffling requires at least two votes in the ballot box. Therefore, the algorithm adds two trivial encryptions (containing an encryption of 1) if there are less than two confirmed votes in the ballot box. The tally control component removes the trivial encryptions during the **ProcessPlaintexts** algorithm. Therefore, the number of actual, mixable votes \hat{N}_C is at least two.

Algorithm 6.1 GetMixnetInitialCiphertexts

Context:

- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$
- Number of eligible voters for this verification card set $N_E \in \mathbb{N}^+$
- Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\text{sup}}] \triangleright$ Can be derived from **pTable** using algorithm 3.9
- Election public key $\mathbf{EL}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\text{max}}}$

Input:

Key-value map of verification card IDs to encrypted, confirmed votes $\mathbf{vcMap}_j = ((\mathbf{vc}_1, \mathbf{E1}_{j,1}) \dots, (\mathbf{vc}_{N_C-1}, \mathbf{E1}_{j,N_C-1})) \in ((\mathbb{A}_{\text{Base16}})^{1_{\text{ID}}} \times \mathbb{G}_q^{\delta+1})^{N_C}$

Require: $N_E \geq N_C$

Operation:

- 1: $\mathbf{vcMap}_{j,\text{ordered}} \leftarrow \text{Order}(\mathbf{vcMap}_j, 1) \triangleright$ Lexicographic ordering by the table's first column (verification card ID)
 - 2: $\mathbf{c}_{\text{init},j} \leftarrow$ Retrieve the encrypted votes from the $\mathbf{vcMap}_{j,\text{ordered}}$.
 - 3: **if** $N_C < 2$ **then** \triangleright We add trivial encryptions to the ballot box, if there are less than 2 votes
 - 4: $\vec{1} \leftarrow \underbrace{(1, \dots, 1)}_{\delta \text{ times}}$
 - 5: $\mathbf{E}_{\text{trivial}} \leftarrow \text{GetCiphertext}(\vec{1}, 1, \mathbf{EL}_{\text{pk}}) \triangleright$ See crypto primitives specification
 - 6: $\mathbf{c}_{\text{init},j} \leftarrow (\mathbf{c}_{\text{init},j}, \mathbf{E}_{\text{trivial}}, \mathbf{E}_{\text{trivial}}) \triangleright$ Adding two trivial encryptions to $\mathbf{c}_{\text{init},j}$
 - 7: **end if**
 - 8: $\mathbf{hvc}_j \leftarrow \text{Base64Encode}(\text{RecursiveHash}(\mathbf{vcMap}_{j,\text{ordered}})) \triangleright$ See crypto primitives specification
-

Output:

CCM_j hash of the encrypted, confirmed votes $\mathbf{hvc}_j \in \mathbb{A}_{\text{Base64}}^{1_{\text{HB64}}}$
 Mix net initial ciphertexts $\mathbf{c}_{\text{init},j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_C}$

Test values for the algorithm 6.1 are provided in [get-mixnet-initial-ciphertexts.json](#).

6.1.2 VerifyMixDecOnline

The online control component verifies the preceding control components' mixing and decryption proofs. The control component bases its verification upon the mix net initial ciphertexts established in the algorithm `GetMixnetInitialCiphertexts`. The first control component $j = 1$ omits the algorithm `VerifyMixDecOnline`.

Algorithm 6.2 VerifyMixDecOnline

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Control component index $j \in [2, 4]$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Ballot box ID $\mathbf{bb} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{sup}] \triangleright$ Can be derived from `pTable` using algorithm 3.9
 Election public key $\mathbf{EL}_{pk} \in \mathbb{G}_q^{\delta_{max}}$
 CCM election public keys $(\mathbf{EL}_{pk,1}, \mathbf{EL}_{pk,2}, \mathbf{EL}_{pk,3}, \mathbf{EL}_{pk,4}) \in (\mathbb{G}_q^{\delta_{max}})^4$
 Electoral board public key $\mathbf{EB}_{pk} \in \mathbb{G}_q^{\delta_{max}}$

Input:

Mix net initial ciphertexts $\mathbf{c}_{init,j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c} \triangleright$ From internal view
 Preceding shuffled votes $\{\mathbf{c}_{mix,k}\}_{k=1}^{j-1} \in ((\mathbb{G}_q^{\delta+1})^{\hat{N}_c})^{j-1}$
 Preceding shuffle proofs $\{\pi_{mix,k}\}_{k=1}^{j-1} \triangleright$ See the domain of the shuffle argument in the crypto primitives specification
 Preceding partially decrypted votes $\{\mathbf{c}_{dec,k}\}_{k=1}^{j-1} \in ((\mathbb{G}_q^{\delta+1})^{\hat{N}_c})^{j-1}$
 Preceding decryption proofs $\{\pi_{dec,k}\}_{k=1}^{j-1} \in ((\mathbb{Z}_q \times \mathbb{Z}_q^{\delta})^{\hat{N}_c})^{j-1}$

Require: $\hat{N}_c \geq 2$

\triangleright The algorithm runs with at least two votes

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: shuffleVerif1 ← VerifyShuffle( $\mathbf{c}_{init,j}$ ,  $\mathbf{c}_{mix,1}$ ,  $\pi_{mix,1}$ ,  $\mathbf{EL}_{pk}$ )
2:  $\mathbf{i}_{aux,1} \leftarrow (\mathbf{ee}, \mathbf{bb}, \text{"MixDecOnline"}, \text{IntegerToString}(1))$ 
3: decryptVerif1 ← VerifyDecryptions( $\mathbf{c}_{mix,1}$ ,  $\mathbf{EL}_{pk,1}$ ,  $\mathbf{c}_{dec,1}$ ,  $\pi_{dec,1}$ ,  $\mathbf{i}_{aux,1}$ )
4: for  $k \in [2, j]$  do  $\triangleright$  We omit the loop for  $j = 2$ 
5:    $\overline{\mathbf{EL}}_{pk} \leftarrow \text{CombinePublicKeys}((\mathbf{EL}_{pk,k}, \dots, \mathbf{EL}_{pk,4}, \mathbf{EB}_{pk}))$ 
6:   shuffleVerifk ← VerifyShuffle( $\mathbf{c}_{dec,k-1}$ ,  $\mathbf{c}_{mix,k}$ ,  $\pi_{mix,k}$ ,  $\overline{\mathbf{EL}}_{pk}$ )
7:    $\mathbf{i}_{aux,k} \leftarrow (\mathbf{ee}, \mathbf{bb}, \text{"MixDecOnline"}, \text{IntegerToString}(k))$ 
8:   decryptVerifk ← VerifyDecryptions( $\mathbf{c}_{mix,k}$ ,  $\mathbf{EL}_{pk,k}$ ,  $\mathbf{c}_{dec,k}$ ,  $\pi_{dec,k}$ ,  $\mathbf{i}_{aux,k}$ )
9: end for
10: if (decryptVerifk  $\wedge$  shuffleVerifk)  $\forall k \in [1, j]$  then
11:   return  $\top$ 
12: else
13:   return  $\perp$ 
14: end if
```

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

6.1.3 MixDecOnline

The online Mixing control component CCM shuffle (and re-encrypt) the previous control component's ciphertexts and perform partial decryption. If the control component is the first to mix ($j = 1$), the mixing public key corresponds to the election public key $\mathbf{EL}_{\mathbf{pk}}$. Each control component ensures that its view of the initial ciphertexts is consistent with the other control components before initiating the mixing process. This is confirmed by validating that the hash of the encrypted, confirmed votes—as computed by each control component in algorithm 6.1—is identical across all control components. In case the hash values are not equal, indicating disagreement among the control components on the list of initial ciphertexts, the process halts. In order to proceed and manage these discrepancies, a third-party could run the dispute resolver, as outlined in section 6.2.

Algorithm 6.3 MixDecOnline

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Control component index $j \in [1, 4]$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Ballot box ID $\mathbf{bb} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{sup}]$ \triangleright Can be derived from pTable using algorithm 3.9
 CCM election public keys $(\mathbf{EL}_{pk,1}, \mathbf{EL}_{pk,2}, \mathbf{EL}_{pk,3}, \mathbf{EL}_{pk,4}) \in (\mathbb{G}_q^{\delta_{max}})^4$
 Electoral board public key $\mathbf{EB}_{pk} \in \mathbb{G}_q^{\delta_{max}}$

Stateful Lists and Maps:

List of \mathbf{bb} of the shuffled and decrypted ballot boxes $\mathbf{L}_{bb,j}$

Input:

Partially decrypted votes $\mathbf{c}_{dec,j-1} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$ \triangleright CCM₁ uses $\mathbf{c}_{init,1}$ from internal view
 CCM_j election secret key $\mathbf{EL}_{sk,j} \in \mathbb{Z}_q^{\delta_{max}}$
 CCM_j hash of the encrypted, confirmed votes $\mathbf{hvc}_j \in \mathbb{A}_{Base64}^{1_{HB64}}$ \triangleright From internal view
 CCM hashes of the encrypted, confirmed votes $\mathbf{hvc} = (\mathbf{hvc}_1, \mathbf{hvc}_2, \mathbf{hvc}_3, \mathbf{hvc}_4) \in (\mathbb{A}_{Base64}^{1_{HB64}})^4$

Require: $\mathbf{hvc}_j = \mathbf{hvc}_1 = \mathbf{hvc}_2 = \mathbf{hvc}_3 = \mathbf{hvc}_4$ \triangleright The view of the initial ciphertexts must be the same for all CCs before mixing begins

Require: $\hat{N}_c \geq 2$

\triangleright The algorithm runs with at least two votes

Require: $\mathbf{bb} \notin \mathbf{L}_{bb,j}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $\overline{\mathbf{EL}}_{pk} \leftarrow \text{CombinePublicKeys}((\mathbf{EL}_{pk,j}, \dots, \mathbf{EL}_{pk,4}, \mathbf{EB}_{pk}))$
- 2: $\mathbf{i}_{aux} \leftarrow (\mathbf{ee}, \mathbf{bb}, \text{"MixDecOnline"}, \text{IntegerToString}(j))$
- 3: $(\mathbf{c}_{mix,j}, \pi_{mix,j}) \leftarrow \text{GenVerifiableShuffle}(\mathbf{c}_{dec,j-1}, \overline{\mathbf{EL}}_{pk})$
- 4: $(\mathbf{c}_{dec,j}, \pi_{dec,j}) \leftarrow \text{GenVerifiableDecryptions}(\mathbf{c}_{mix,j}, (\mathbf{EL}_{pk,j}, \mathbf{EL}_{sk,j}), \mathbf{i}_{aux})$
- 5: $\mathbf{L}_{bb,j} \leftarrow \mathbf{L}_{bb,j} \cup \mathbf{bb} \quad \mathbf{L}_{bb,j} \leftarrow \mathbf{L}_{bb,j} \parallel \mathbf{bb}$

Output:

Shuffled votes $\mathbf{c}_{mix,j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
 Shuffle proof $\pi_{mix,j}$ \triangleright See the domain of the shuffle argument in the crypto primitives specification
 Partially decrypted votes $\mathbf{c}_{dec,j} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
 Decryption proofs $\pi_{dec,j} \in (\mathbb{Z}_q \times \mathbb{Z}_q^{\delta})^{\hat{N}_c}$

6.1.4 ~~Handling Inconsistent Views of Confirmed Votes~~

6.2 Handling Inconsistent Views of Confirmed Votes

~~Section 6.1.1 highlights that each control component independently retrieves the list of confirmed votes in the algorithm. Before initiating the mixing process in the algorithm, each control component verifies the consistency of the lists of confirmed votes across all control components.~~

6.2.1 Overview and Trigger of the Dispute Resolution Process

This section elaborates on the scenario ~~, where in which~~ the control components do not agree on the list of confirmed votes. The Federal Chancellery's Ordinance and the explanatory report require to anticipate inconsistencies between control components.

[VEleS Annex 11.11]: The canton anticipates any anomalies and, in consultation with the bodies concerned, draws up an emergency plan specifying the appropriate course of action. It creates transparency towards the public.

[Explanatory Report, Sec 4.2.1]: As a condition for the successful examination of the proof referred to in Number 2.6, all control components must have recorded the same votes as having been cast in conformity with the system. Cases where the control components show inconsistencies in this respect must be anticipated in accordance with Number 11.11 and the procedure determined in advance.

In this section, we assume a ~~trustworthy~~ *dispute resolver* (see section 2.10): a party outside the cryptographic protocol who can interact with the control components and can ~~check~~ verify the authenticity of the protocol ~~'s~~ messages (see section 7).

~~The algorithms below highlight the operations that the dispute resolver must run to unambiguously resolve the inconsistencies. However, we highlight that the Swiss Post Voting System does not aim to provide accountability: the property whereby a misbehaving entity can be identified.~~ Section 6.1.1 describes how each control component independently retrieves its list of confirmed votes using the GetMixnetInitialCiphertexts algorithm. Before the mixing process begins in the MixDecOnline algorithm, each control component verifies that the list of confirmed votes is consistent across all control components.

~~If the control components have a different view of the confirmed votes, the authorities would launch a thorough forensic investigation; the details of which are out of scope of this document.~~ In exceptional cases, discrepancies may arise between control components regarding the list of confirmed votes. In such situations, the cantons require the intervention of a dispute resolver to resolve the inconsistency.

Figure 15 outlines the sequence of operations that the dispute resolver performs to unambiguously identify and resolve inconsistencies.

~~The control components are required to retain all messages received from other parties for potential future forensic analysis.~~

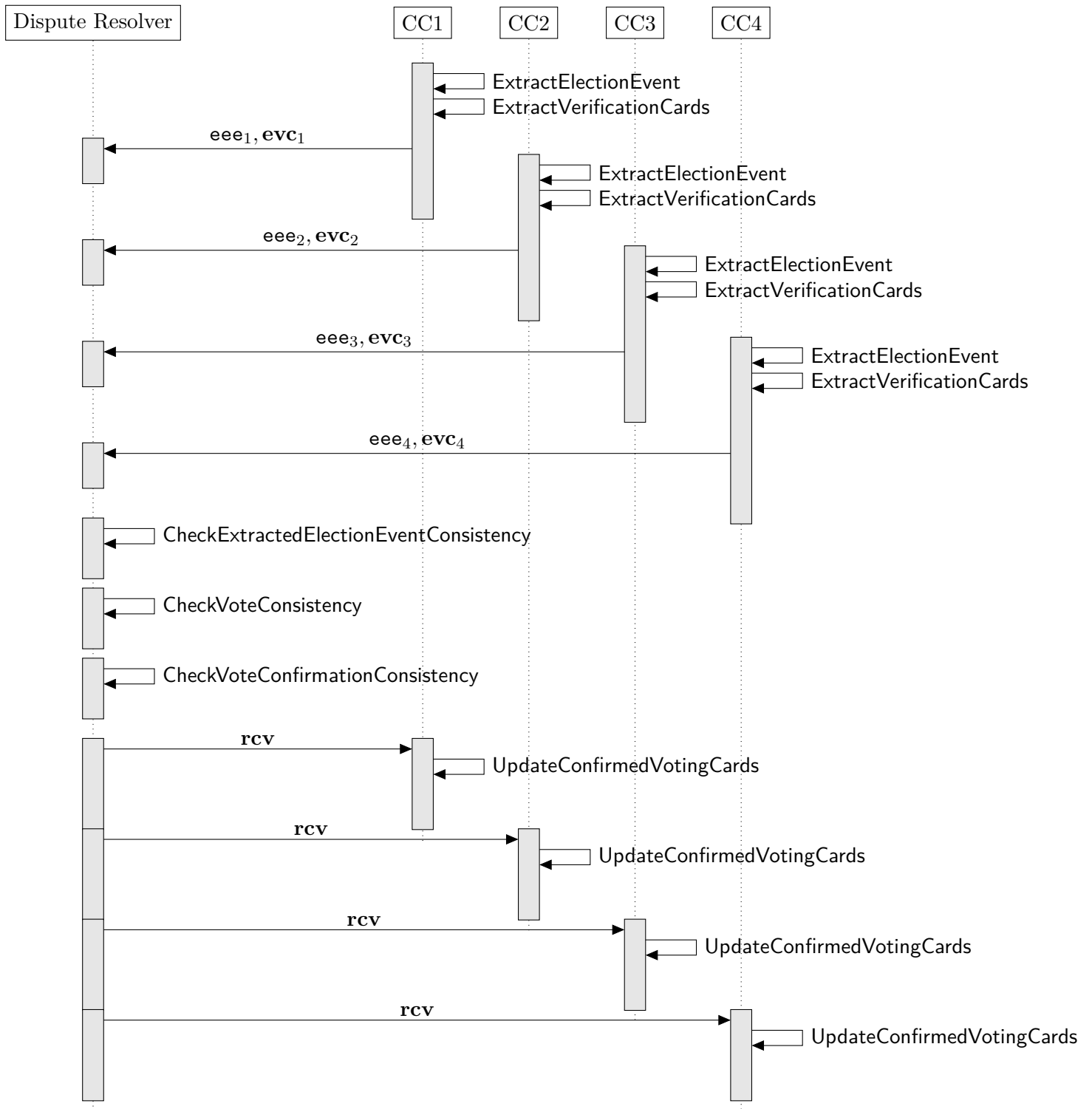


Fig. 15: [Sequence diagram of the DisputeResolver protocol.](#)

We distinguish two types of inconsistencies

6.2.2 Extraction Phase

The dispute resolution process begins by requesting each control component to invoke the following algorithms:

- Different ciphertexts E_1 for a given verification card ID vc_{id} . This also includes the case where a control component did not register a ciphertext E_1 at all for this verification card ID vc_{id} . [3.13 ExtractElectionEvent](#)
- A different confirmation status of a given verification card ID. Here, we assume that the control components registered the same ciphertext E_1 , but disagree whether the vote was confirmed or not. [3.16 ExtractVerificationCards](#)

The first algorithm resolves the first type of inconsistencies by re-executing parts of the SendVote protocol while the ConfirmVoteAgreement algorithm resolves the second type of inconsistencies by re-executing parts of the ConfirmVote protocol. Each control component securely transfers its extracted election event and associated verification cards to the dispute resolver via an out-of-band channel (see table 17).

We assume that the *dispute resolver* executes for every inconsistent view of each contested verification card ID vc_{id} . Since at least one control component is trustworthy, we know that only one execution of can successfully verify. As a result of a successful execution, the corresponding ciphertext E_1 must be used for the tally phase. Prior to executing the algorithms, the *dispute resolver* checks the authenticity and validity of the messages (see section 7)

6.2.3 Dispute Resolver's Consistency Checks

The dispute resolver receives, as input, the extracted election event and associated verification cards from each control component.

Before proceeding with the resolution protocol, the dispute resolver checks the authenticity and validity of the messages (see section 7). Moreover, the operator of the dispute resolver manually checks that the received inputs correspond to the expected election event. The dispute resolver then executes the following algorithms:

- [6.4 CheckExtractedElectionEventConsistency](#)
- [6.5 CheckVoteConsistency](#)
- [6.6 CheckVoteConfirmationConsistency](#)

The [CheckExtractedElectionEventConsistency](#) algorithm checks that all control components agree on the election event context. This shared context is used by the dispute resolver in subsequent verifications. If the context is not identical across all control components, the algorithm fails.

Algorithm 6.4 CheckExtractedElectionEventConsistency

Context:

~~Group modulus $p \in \mathbb{P}$ Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ Group generator $g \in \mathbb{G}_q$ Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$ Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1_{ID}}$ Verification card ID $vc_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$ Election public key $EL_{pk} = (EL_{pk,0}, \dots, EL_{pk,\delta_{max}-1}) \in \mathbb{G}_q^{\delta_{max}}$ The CCR's Extracted election event $(eee_1, eee_2, eee_3, eee_4)$~~

$\triangleright eee = (hContext, (p, q, g), ee, evcs)$

Operation:

```

1: for  $j \in [1, 4]$  do
2:    $h_{eee_j} \leftarrow \text{GetHashExtractedElectionEvent}()$   $\triangleright$  See algorithm 3.15
3: end for
4: if  $h_{eee_1} = h_{eee_2} = h_{eee_3} = h_{eee_4}$  then
5:   return  $\top$ 
6: else
7:   return  $\perp$ 
8: end if

```

Output:

~~Choice Return Codes encryption public key $pk_{CCR} \in \mathbb{G}_q^{n_{max}}$ The result of the algorithm: \top if the context is identical, \perp otherwise.~~

Next, the dispute resolver runs the CheckVoteConsistency algorithm to ensure that all control components have the same view of the submitted, encrypted votes. If discrepancies are found, the algorithm fails.

Algorithm 6.5 CheckVoteConsistency

Input:

~~The CCR's exponentiated elements $(\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4) \in (\mathbb{G}_q^\psi)^4$ The CCR's exponentiation proofs $(\pi_{\text{decPCC},1}, \pi_{\text{decPCC},2}, \pi_{\text{decPCC},3}, \pi_{\text{decPCC},4}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi \times 4}$ The CCR's Choice Return Codes encryption keys $(\mathbf{pk}_{\text{CCR},1}, \mathbf{pk}_{\text{CCR},2}, \mathbf{pk}_{\text{CCR},3}, \mathbf{pk}_{\text{CCR},4}) \in (\mathbb{G}_q^{\psi_{\text{max}}})^4$ Encrypted vote $\mathbf{E1} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1}) \in \mathbb{G}_q^{\delta+1}$ Exponentiated encrypted vote $\widetilde{\mathbf{E1}} = (\gamma_1^{k_{\text{id}}}, \phi_{1,0}^{k_{\text{id}}}) \in \mathbb{G}_q^2$ Encrypted partial Choice Return Codes $\mathbf{E2} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1}) \in \mathbb{G}_q^{\psi+1}$ CCR's extracted verification cards $(\mathbf{evc}_1, \mathbf{evc}_2, \mathbf{evc}_3, \mathbf{evc}_4)$~~
 $\triangleright \mathbf{evc} = (\mathbf{evc}_0, \dots, \mathbf{evc}_{N_S-1})$, see algorithm 3.16 for the content

Operation: $\mathbf{i}_{\text{aux}} \leftarrow (\text{"PartialDecryptPCC", } \mathbf{vc}_{\text{id}}, \text{GetHashContext}())$

$\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$

$\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1^{k_{\text{id}}}), \text{IntegerToString}(\phi_{1,0}^{k_{\text{id}}}))$ $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}))$

$\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(j))$

\triangleright We use nested structures in this algorithm

1: **for** $j \in [1, 4]$ **do**

2: **for** $i \in [0, N_S)$ **do**

3: ~~$\mathbf{pk}_{\text{CCR},j} = (\mathbf{pk}_{\text{CCR},j,0}, \dots, \mathbf{pk}_{\text{CCR},j,\psi_{\text{max}}-1})$~~ $h_{\mathbf{E1},j,i} \leftarrow (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1})$

4: ~~$\mathbf{d}_j = (d_{j,0}, \dots, d_{j,\psi-1})$~~ $h_{\mathbf{evc},j,i} \leftarrow (\mathbf{vc}_{\text{id},j,i}, \mathbf{vcs}_{j,i}, h_{\mathbf{E1},j,i})$

5: **end for**

6: ~~$\pi_{\text{decPCC},j} = (\pi_{\text{decPCC},j,0}, \dots, \pi_{\text{decPCC},j,\psi-1})$~~ $h_j \leftarrow (h_{\mathbf{evc},j,0}, \dots, h_{\mathbf{evc},j,N_S-1})$

7: ~~$\text{ExpVerif}_{j,i} \leftarrow \text{VerifyExponentiation}((g, \gamma_2), (\mathbf{pk}_{\text{CCR},j,i}, d_{j,i}), \pi_{\text{decPCC},j,i}, \mathbf{i}_{\text{aux}})$~~

$d_j \leftarrow \text{Base64Encode}(\text{RecursiveHash}(h_j))$

\triangleright See crypto primitives specification

8: **end for**

9: **if** $d_1 = d_2 = d_3 = d_4$ **then**

10: **return** \top

11: **else**

12: **return** \perp

13: **end if**

Output:

The result of the algorithm: \top if the ~~encrypted vote should be part of the tally (if it was confirmed)~~ list of sent votes is identical, \perp otherwise.

~~After the algorithm, the~~ The Swiss Post Voting System enforces consistency among control components with respect to both the election event context and the encrypted votes throughout the voting phase. This ensures that a misbehaving entity can be identified. All control components must ~~agree on the content~~ have used the same context and encrypted votes during the voting phase, as they commit to these values in PartialDecryptPCC and verify each other's zero-knowledge proofs in DecryptPCC. The zero-knowledge proofs generated during the voting phase include the ExtractElectionEvent and the hash of the encrypted votes. ~~However, they still might disagree whether a vote was correctly confirmed. Therefore, the dispute resolver executes ConfirmVoteAgreement for every vote in the auxiliary information, in analogy to what is extracted for the dispute resolver (see section 6.2.2).~~

Thus, the SendVote protocol establishes an explicit agreement on the election event context and the encrypted votes.

However, the Swiss Post Voting System does not provide a similar agreement mechanism for the vote confirmation status during the voting phase. Thus, discrepancies in vote confirmation status may occur, that is, cases where the control components for a given verification card ID vc_{id} with a contested confirmation status. If the ~~ConfirmVoteAgreement~~ runs successfully for at least one view, registered the same ciphertext, but disagree whether the vote was confirmed or not. Such discrepancies can only be resolved through the dispute resolution process. To this end, the dispute resolver constructs the list of confirmed votes in the CheckVoteConfirmationConsistency algorithm. In this exceptional scenario, the ~~vote must be included in the tally phase.~~

Similarly to the algorithm, we assume that the ~~dispute resolver~~ checks the authenticity and validity of the context and input parameters. ~~control components~~ subsequently update their view of the ballot box in the UpdateConfirmedVotingCards algorithm.

Algorithm 6.6 ~~ConfirmVoteAgreement~~ CheckVoteConfirmationConsistency

Context:

~~Group modulus~~ $p \in \mathbb{P}$ ~~Extracted election event~~ $\text{eee} = (\text{hContext}, (p, q, g), \text{ee}, \text{evcs})$

Input:

~~Group cardinality~~ $q \in \mathbb{P}$ s.t. $p = 2q + 1$ ~~The CCR's extracted verification cards~~
 $(\text{evc}_1, \text{evc}_2, \text{evc}_3, \text{evc}_4)$

$\triangleright \text{evc} = (\text{evc}_0, \dots, \text{evc}_{N_S-1})$, see algorithm 3.16 for the content

Operation:

```

1:  $\text{rcv} \leftarrow ()$ 
2: for  $j \in [1, 4]$  do
3:   for  $i \in [0, N_S)$  do
4:     if  $\text{vc}_{\text{id},j,i} \notin \{\text{vc}_{\text{id},x} \mid \text{rcv}_x \in \text{rcv}\} \wedge \text{evc}_{j,i} \neq (\text{vc}_{\text{id},j,i}, \text{vcs}_{j,i}, \text{E1}_{j,i}, ())$  then
5:        $\text{evc}_{j,i} = (\text{vc}_{\text{id},j,i}, \text{vcs}_{j,i}, \text{E1}_{j,i}, (\text{hlVCC}_{1,\text{id}}, \text{hlVCC}_{2,\text{id}}, \text{hlVCC}_{3,\text{id}}, \text{hlVCC}_{4,\text{id}})_{j,i})$ 
6:        $\text{Verif}_{j,i} \leftarrow \text{ConfirmVoteAgreement}((\text{hlVCC}_{1,\text{id}}, \text{hlVCC}_{2,\text{id}}, \text{hlVCC}_{3,\text{id}}, \text{hlVCC}_{4,\text{id}})_{j,i})$   $\triangleright$  See
algorithm 6.7
7:       if  $\text{Verif}_{j,i}$  then
8:          $\text{rcv} \leftarrow \text{rcv} \parallel (\text{vc}_{\text{id},j,i}, \text{vcs}_{j,i}, (\text{hlVCC}_{1,\text{id}}, \text{hlVCC}_{2,\text{id}}, \text{hlVCC}_{3,\text{id}}, \text{hlVCC}_{4,\text{id}})_{j,i})$ 
9:       end if
10:    end if
11:  end for
12: end for
13:  $\text{rcv} \leftarrow \text{Order}(\text{rcv})$   $\triangleright$  Lexicographic ordering by the verification card id.
```

Output:

~~Group generator~~ $g \in \mathbb{G}_q$ ~~The dispute resolver's resolved confirmed votes~~
 $\text{rcv} = (\text{rcv}_0, \dots, \text{rcv}_{N_S-1})$

Algorithm 6.7 ConfirmVoteAgreement

Context:

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card set ID $vcs \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Verification card ID $vc_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Long Vote Cast Return Codes allow list $L_{lvcc} \in (\mathbb{A}_{Base64}^{1_{HB64}})^{N_E}$

Input:

The	CCR's	hashed	long	Vote	Cast	Return	Code
shares							
							$(hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}) \in (\mathbb{A}_{Base64}^{1_{HB64}})^4$
							$(hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}) \in (\mathbb{A}_{Base64}^{1_{HB64}})^4$

Operation:

- 1: $i_{aux} \leftarrow (\text{"VerifyLVCCHash"}, ee, vcs, vc_{id})$
 - 2: $hh1VCC_{id} \leftarrow \text{Base64Encode}(\text{RecursiveHash}(i_{aux}, hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}))$
 \triangleright See crypto primitives specification
 - 3: **if** $hh1VCC_{id} \in L_{lvcc}$ **then**
 - 4: **return** \top
 - 5: **else**
 - 6: **return** \perp
 - 7: **end if**
-

Output:

The result of the algorithm: \top if the vote should be part of the tally, \perp otherwise.

~~After resolving-~~

6.2.4 Control Component's Update

After the dispute resolver has resolved the inconsistencies, the control components ~~re-run the tally phase with the agreed~~ update their list of confirmed votes. To minimize the impact of the dispute resolution process, updates are restricted to changes from *SENT* to *CONFIRMED* status for individual votes. We prevent the control components from modifying or dropping confirmed votes.

Operational safeguards ensure that this update can only occur with approval from the cantons.

Algorithm 6.8 UpdateConfirmedVotingCards

Context:

The CCR's index $j \in [1, 4]$

Election event ID $ee \in (\mathbb{A}_{Base16})^{1_{ID}}$

Long Vote Cast Return Codes allow lists $(L_{1VCC,0}, \dots, L_{1VCC,N_{bb}-1}) \in ((\mathbb{A}_{Base64}^{1_{HB64}})^{N_E})^{N_{bb}}$

Stateful Lists and Maps:

CCR_j's list of sent voting cards $L_{sentVotes,j}$

CCR_j's list of confirmed ~~votes~~ voting cards $L_{confirmedVotes,j}$

Input:

The dispute resolver's resolved confirmed votes $rcv = (rcv_0, \dots, rcv_{N_c-1})$

Operation:

```

1: for  $(vc_{id}, vcs, (hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id})) \in rcv$  do
2:   if  $vc_{id} \notin L_{sentVotes,j}$  then
3:     return  $\perp$ 
4:   end if
5:   if  $vc_{id} \notin L_{confirmedVotes,j}$  then
6:      $Verif \leftarrow ConfirmVoteAgreement((hlVCC_{1,id}, hlVCC_{2,id}, hlVCC_{3,id}, hlVCC_{4,id}))$   $\triangleright$  See
       algorithm 6.7
7:     if  $Verif$  then
8:        $L_{confirmedVotes,j} \leftarrow L_{confirmedVotes,j} \parallel vc_{id}$ 
9:     else
10:      return  $\perp$ 
11:    end if
12:  end if
13: end for

```

Output:

The result of the algorithm: \top if the update was successful, \perp otherwise.

After a successful execution of the dispute resolution process, the control components re-initialize the tally phase by re-running the *GetMixnetInitialCiphertexts* algorithm. The dispute resolution process is executed only once. In the event that the control components maintain inconsistent views, the canton will initiate a comprehensive forensic investigation (details are beyond the scope of this document). All control components must retain all received messages for potential future analysis.

6.3 MixOffline

6.3.1 VerifyVotingClientProofs

The Tally control component verifies the consistency of the online control components' encrypted confirmed votes $\{\mathbf{vc}_j, \mathbf{E1}_j, \widetilde{\mathbf{E1}}_j, \mathbf{E2}_j, \pi_{\text{Exp},j}, \pi_{\text{EqEnc},j}\}_{j=1}^4$. After this check, the Tally control component proceeds with verifying the voting client's zero-knowledge proofs. By convention, we use the first control component's confirmed encrypted votes.

Algorithm 6.9 VerifyVotingClientProofs

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1\text{ID}}$
 Verification card set ID $\mathbf{vcs} \in (\mathbb{A}_{Base16})^{1\text{ID}}$
 Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n$ \triangleright \mathbf{pTable} is of the form
 $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ \triangleright ψ and δ can be derived from \mathbf{pTable} using algorithms 3.8 and 3.9
 Number of eligible voters for this verification card set $N_E \in \mathbb{N}^+$
 Election public key $\mathbf{EL}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\text{max}}}$
 Choice Return Codes encryption public key $\mathbf{pk}_{\text{CCR}} \in \mathbb{G}_q^{\psi_{\text{max}}}$

Input:

Control component's list of confirmed verification card IDs $\mathbf{vc}_1 = (\mathbf{vc}_{1,0}, \dots, \mathbf{vc}_{1,N_C-1}) \in (\mathbb{A}_{Base16})^{1\text{ID}}^{N_C}$
 Control component's list of encrypted, confirmed votes $\mathbf{E1}_1 = (\mathbf{E1}_{1,0}, \dots, \mathbf{E1}_{1,N_C-1}) \in (\mathbb{G}_q^{\delta+1})^{N_C}$
 Control component's list of exponentiated, encrypted, confirmed votes $\widetilde{\mathbf{E1}}_1 = (\widetilde{\mathbf{E1}}_{1,0}, \dots, \widetilde{\mathbf{E1}}_{1,N_C-1}) \in (\mathbb{G}_q^2)^{N_C}$
 Control component's list of encrypted, partial Choice Return Codes $\mathbf{E2}_1 = (\mathbf{E2}_{1,0}, \dots, \mathbf{E2}_{1,N_C-1}) \in (\mathbb{G}_q^{\psi+1})^{N_C}$
 Control component's list of exponentiation proofs $\pi_{\text{Exp},1} = (\pi_{\text{Exp},1,0}, \dots, \pi_{\text{Exp},1,N_C-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{N_C}$
 Control component's list of plaintext equality proofs $\pi_{\text{EqEnc},1} = (\pi_{\text{EqEnc},1,0}, \dots, \pi_{\text{EqEnc},1,N_C-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q^2)^{N_C}$
 Key-value map of the verification card public keys $\mathbf{KMap} = ((\mathbf{vc}_0, K_0), \dots, (\mathbf{vc}_{N_E-1}, K_{N_E-1})) \in ((\mathbb{A}_{Base16})^{1\text{ID}} \times \mathbb{G}_q)^{N_E}$

Require: $N_E \geq N_C \geq 1$

Require: $\mathbf{vc}_{1,i} \neq \mathbf{vc}_{1,k}, \forall i, k \in \{0, \dots, (N_C - 1)\} \wedge i \neq k$

\triangleright The algorithm runs with at least one confirmed vote
 \triangleright All verification card IDs must be distinct

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1:  $\widetilde{\mathbf{pk}}_{\text{CCR}} \leftarrow \prod_{k=0}^{\psi-1} \mathbf{pk}_{\text{CCR},i} \bmod p$ 
2: for  $i \in [0, N_C)$  do
3:    $K_{\text{id}} \leftarrow \mathbf{KMap}(\mathbf{vc}_{1,i})$   $\triangleright$  Retrieve the value from the key-value map
4:    $\mathbf{E1}_{1,i} = (\gamma_1, \phi_{1,0}, \dots, \phi_{1,\delta-1})$ 
5:    $\widetilde{\mathbf{E1}}_{1,i} = (\gamma_1^{K_{\text{id}}}, \phi_{1,0}^{K_{\text{id}}})$ 
6:    $\mathbf{E2}_{1,i} = (\gamma_2, \phi_{2,0}, \dots, \phi_{2,\psi-1})$ 
7:    $\widetilde{\mathbf{E2}}_i \leftarrow (\gamma_2, \prod_{k=0}^{\psi-1} \phi_{2,k} \bmod p)$ 
8:    $\mathbf{i}_{\text{aux}} \leftarrow (\text{"CreateVote"}, \mathbf{vc}_{1,i}, \text{GetHashContext}())$   $\triangleright$  See algorithm 3.12
9:    $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_1), \text{IntegerToString}(\phi_{1,0}), \dots, \text{IntegerToString}(\phi_{1,\delta-1}))$ 
10:   $\mathbf{i}_{\text{aux}} \leftarrow (\mathbf{i}_{\text{aux}}, \text{IntegerToString}(\gamma_2), \text{IntegerToString}(\phi_{2,0}), \dots, \text{IntegerToString}(\phi_{2,\psi-1}))$ 
11:   $\text{VerifExp}_i \leftarrow \text{VerifyExponentiation}((g, \gamma_1, \phi_{1,0}), (K_{\text{id}}, \gamma_1^{K_{\text{id}}}, \phi_{1,0}^{K_{\text{id}}}), \pi_{\text{Exp},1,i}, \mathbf{i}_{\text{aux}})$ 
12:   $\text{VerifEqEnc}_i \leftarrow \text{VerifyPlaintextEquality}(\widetilde{\mathbf{E1}}_{1,i}, \widetilde{\mathbf{E2}}_i, \mathbf{EL}_{\text{pk},0}, \widetilde{\mathbf{pk}}_{\text{CCR}}, \pi_{\text{EqEnc},1,i}, \mathbf{i}_{\text{aux}})$ 
13: end for
14: if  $(\text{VerifExp}_i \wedge \text{VerifEqEnc}_i) \forall i \in [0, N_C)$  then
15:   return  $\top$ 
16: else
17:   return  $\perp$ 
18: end if
```

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

6.3.2 VerifyMixDecOffline

After the verification of the voting client's zero-knowledge proofs, the Tally control component verifies the online control components' shuffle and decryption proofs. Using the algorithm `GetMixnetInitialCiphertexts`, the Tally control component retrieves the first online control component's mixnet initial ciphertexts $\mathbf{c}_{\text{init},1}$ from its encrypted confirmed votes (the Tally control component verifies the consistency of the control components' confirmed encrypted votes prior to the execution of this algorithm).

Algorithm 6.10 VerifyMixDecOffline

Context:

- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$
- Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{\text{ID}}}$
- Ballot box ID $\mathbf{bb} \in (\mathbb{A}_{Base16})^{1_{\text{ID}}}$
- Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{\text{sup}}] \triangleright$ Can be derived from `pTable` using algorithm 3.9
- Election public key $\mathbf{EL}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\text{max}}}$
- CCM election public keys $(\mathbf{EL}_{\text{pk},1}, \mathbf{EL}_{\text{pk},2}, \mathbf{EL}_{\text{pk},3}, \mathbf{EL}_{\text{pk},4}) \in (\mathbb{G}_q^{\delta_{\text{max}}})^4$
- Electoral board public key $\mathbf{EB}_{\text{pk}} \in \mathbb{G}_q^{\delta_{\text{max}}}$

Input:

- Mix net initial ciphertexts $\mathbf{c}_{\text{init},1} \in (\mathbb{G}_q^{\delta+1})^{\hat{\mathbf{N}}_{\text{c}}}$
- Preceding shuffled votes $\{\mathbf{c}_{\text{mix},j}\}_{j=1}^4 \in ((\mathbb{G}_q^{\delta+1})^{\hat{\mathbf{N}}_{\text{c}}})^4$
- Preceding shuffle proofs $\{\pi_{\text{mix},j}\}_{j=1}^4 \triangleright$ See the domain of the shuffle argument in the crypto primitives specification
- Preceding partially decrypted votes $\{\mathbf{c}_{\text{dec},j}\}_{j=1}^4 \in ((\mathbb{G}_q^{\delta+1})^{\hat{\mathbf{N}}_{\text{c}}})^4$
- Preceding decryption proofs $\{\pi_{\text{dec},j}\}_{j=1}^4 \in ((\mathbb{Z}_q \times \mathbb{Z}_q^{\delta})^{\hat{\mathbf{N}}_{\text{c}}})^4$

Require: $\hat{\mathbf{N}}_{\text{c}} \geq 2$

\triangleright The algorithm runs with at least two votes

Operation:

\triangleright For all algorithms see the crypto primitives specification

- 1: $\text{shuffleVerif}_1 \leftarrow \text{VerifyShuffle}(\mathbf{c}_{\text{init},1}, \mathbf{c}_{\text{mix},1}, \pi_{\text{mix},1}, \mathbf{EL}_{\text{pk}})$
- 2: $\mathbf{i}_{\text{aux},1} \leftarrow (\mathbf{ee}, \mathbf{bb}, \text{"MixDecOnline"}, \text{IntegerToString}(1))$
- 3: $\text{decryptVerif}_1 \leftarrow \text{VerifyDecryptions}(\mathbf{c}_{\text{mix},1}, \mathbf{EL}_{\text{pk},1}, \mathbf{c}_{\text{dec},1}, \pi_{\text{dec},1}, \mathbf{i}_{\text{aux},1})$
- 4: **for** $j \in [2, 4]$ **do**
- 5: $\overline{\mathbf{EL}}_{\text{pk}} \leftarrow \text{CombinePublicKeys}((\mathbf{EL}_{\text{pk},j}, \dots, \mathbf{EL}_{\text{pk},4}, \mathbf{EB}_{\text{pk}}))$
- 6: $\text{shuffleVerif}_j \leftarrow \text{VerifyShuffle}(\mathbf{c}_{\text{dec},j-1}, \mathbf{c}_{\text{mix},j}, \pi_{\text{mix},j}, \overline{\mathbf{EL}}_{\text{pk}})$
- 7: $\mathbf{i}_{\text{aux},j} \leftarrow (\mathbf{ee}, \mathbf{bb}, \text{"MixDecOnline"}, \text{IntegerToString}(j))$
- 8: $\text{decryptVerif}_j \leftarrow \text{VerifyDecryptions}(\mathbf{c}_{\text{mix},j}, \mathbf{EL}_{\text{pk},j}, \mathbf{c}_{\text{dec},j}, \pi_{\text{dec},j}, \mathbf{i}_{\text{aux},j})$
- 9: **end for**
- 10: **if** $(\text{decryptVerif}_j \wedge \text{shuffleVerif}_j) \forall j \in [1, 4]$ **then**
- 11: **return** \top
- 12: **else**
- 13: **return** \perp
- 14: **end if**

Output:

The result of the verification: \top if the verification is successful, \perp otherwise.

6.3.3 MixDecOffline

If all verifications in the `VerifyMixDecOffline` algorithm succeed, the Tally control component interacts with the electoral board (see section 2.5) and shuffles (and re-encrypts) the votes and performs the final decryption.

Algorithm 6.11 MixDecOffline

Context:

Group modulus $p \in \mathbb{P}$
 Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
 Group generator $g \in \mathbb{G}_q$
 Election event ID $\mathbf{ee} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Ballot box ID $\mathbf{bb} \in (\mathbb{A}_{Base16})^{1_{ID}}$
 Number of allowed write-ins + 1 for this specific ballot box $\delta \in [1, \delta_{sup}]$ \triangleright Can be derived from pTable using algorithm 3.9

Stateful Lists and Maps:

List of \mathbf{bb} of the shuffled and decrypted ballot boxes $L_{\mathbf{bb}, \text{Tally}}$

Input:

Partially decrypted votes $\mathbf{c}_{dec,4} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
 Passwords of k electoral board members $(PW_0, \dots, PW_{k-1}) \in (\mathbb{A}_{UCS^*})^k$

Require: $\hat{N}_c \geq 2$

\triangleright The algorithm runs with at least two votes

Require: $k \geq 2$

\triangleright We require at least 2 electoral board members

Require: $\mathbf{bb} \notin L_{\mathbf{bb}, \text{Tally}}$

Operation:

\triangleright For all algorithms see the crypto primitives specification

```

1: for  $i \in [0, \delta)$  do
2:    $EB_{sk,i} \leftarrow \text{RecursiveHashToZq}(q, ("ElectoralBoardSecretKey", \mathbf{ee}, i, PW_0, \dots, PW_{k-1}))$ 
3:    $EB_{pk,i} \leftarrow g^{EB_{sk,i}} \bmod p$ 
4: end for
5:  $EB_{sk} \leftarrow (EB_{sk,0}, \dots, EB_{sk,\delta-1})$ 
6:  $EB_{pk} \leftarrow (EB_{pk,0}, \dots, EB_{pk,\delta-1})$ 
7:  $i_{aux} \leftarrow (\mathbf{ee}, \mathbf{bb}, "MixDecOffline")$ 
8:  $(\mathbf{c}_{mix,5}, \pi_{mix,5}) \leftarrow \text{GenVerifiableShuffle}(\mathbf{c}_{dec,4}, EB_{pk})$ 
9:  $(\mathbf{c}_{dec,5}, \pi_{dec,5}) \leftarrow \text{GenVerifiableDecryptions}(\mathbf{c}_{mix,5}, (EB_{pk}, EB_{sk}), i_{aux})$ 
10: for  $i \in [0, \hat{N}_c)$  do
11:    $\mathbf{c}_{dec,5,i} = (\gamma_i, \phi_{i,0}, \dots, \phi_{i,\delta-1})$ 
12:    $m_i \leftarrow (\phi_{i,0}, \dots, \phi_{i,\delta-1})$ 
13: end for
14:  $\mathbf{m} \leftarrow (m_0, \dots, m_{\hat{N}_c-1})$ 
15:  $L_{\mathbf{bb}, \text{Tally}} \leftarrow L_{\mathbf{bb}, \text{Tally}} \cup \mathbf{bb}$   $L_{\mathbf{bb}, \text{Tally}} \leftarrow L_{\mathbf{bb}, \text{Tally}} || \mathbf{bb}$ 

```

Output:

Shuffled votes $\mathbf{c}_{mix,5} \in (\mathbb{G}_q^{\delta+1})^{\hat{N}_c}$
 Shuffle proof $\pi_{mix,5}$ \triangleright See the domain of the shuffle argument in the crypto primitives specification
 Decrypted votes $\mathbf{m} = (m_0, \dots, m_{\hat{N}_c-1}) \in (\mathbb{G}_q^{\delta})^{\hat{N}_c}$
 Decryption proofs $\pi_{dec,5} \in (\mathbb{Z}_q \times \mathbb{Z}_q^{\delta})^{\hat{N}_c}$

6.3.4 ProcessPlaintexts

m contains the list of plaintext votes; each message m consists of the multiplied primes $\hat{\mathbf{p}} = (\hat{p}_0, \dots, \hat{p}_{\psi-1})$ encoding the selected ~~voting options~~ $(v_0, \dots, v_{\psi-1})$ actual voting options $(\hat{v}_0, \dots, \hat{v}_{\psi-1})$. The Tally control component retrieves the list of encoded voting options $\tilde{\mathbf{p}}$ from the primes mapping table **pTable**, factorizes each plaintext vote to retrieve the voter's encoded voting options and decodes them to the actual voting options. If the original ballot box contained less than two confirmed votes ($N_c < 2$), the **ProcessPlaintexts** algorithm strips away the trivial encryptions that the **GetMixnetInitialCiphertexts** algorithm added. Furthermore, if write-ins are present in the encrypted vote, the algorithm decodes the write-in fields, if they are present (see section 3.8.5)

Algorithm 6.12 ProcessPlaintexts

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

Primes mapping table $\mathbf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}^* \times \mathcal{T}_1^{50})^n \triangleright \mathbf{pTable}$ is of the form $((v_0, \tilde{p}_0, \sigma_0, \tau_0), \dots, (v_{n-1}, \tilde{p}_{n-1}, \sigma_{n-1}, \tau_{n-1})) \triangleright \psi, \delta, \tilde{\mathbf{p}}$ and $\tilde{\mathbf{p}}_w$ can be derived from \mathbf{pTable} using algorithms 3.8, 3.9, 3.2 and 3.7

Input:

List of plaintext votes $\mathbf{m} = (m_0, \dots, m_{\hat{N}_c-1}) \in (\mathbb{G}_q^\delta)^{\hat{N}_c}$

Require: $\hat{N}_c \geq 2$

\triangleright The algorithm runs with at least two votes

Operation:

```

1:  $L_{\text{votes}} \leftarrow ()$ 
2:  $L_{\text{decodedVotes}} \leftarrow ()$ 
3:  $L_{\text{writeIns}} \leftarrow ()$ 
4:  $\vec{1} \leftarrow \underbrace{(1, \dots, 1)}_{\delta \text{ times}}$ 
5:  $\hat{\tau} \leftarrow \text{GetBlankCorrectnessInformation}()$   $\triangleright$  See Algorithm 3.6
6:  $k \leftarrow 0$ 
7: for  $i \in [0, \hat{N}_c)$  do
8:    $m_i = (\phi_{i,0}, \dots, \phi_{i,\delta-1})$ 
9:   if  $m_i \neq \vec{1}$  then
10:     $\hat{\mathbf{p}}_k \leftarrow \text{Factorize}(\phi_{i,0})$   $\triangleright$  See Algorithm 3.10
11:     $\hat{\mathbf{v}}_k \leftarrow \text{GetActualVotingOptions}(\hat{\mathbf{p}}_k)$   $\triangleright$  See Algorithm 3.3
12:     $\tau' \leftarrow \text{GetCorrectnessInformation}(\hat{\mathbf{v}}_k)$   $\triangleright$  See Algorithm 3.5
13:    if  $\hat{\tau} \neq \tau'$  then
14:      return  $\perp$   $\triangleright$  If the vote contains an invalid combination of voting options (see section 3.5.4), the algorithm aborts and returns nothing
15:    end if
16:     $\mathbf{w}_k \leftarrow (\phi_{i,1}, \dots, \phi_{i,\delta-1})$   $\triangleright$  An empty vector if the election event has no write-ins
17:     $\hat{\mathbf{s}}_k \leftarrow \text{DecodeWriteIns}(\hat{\mathbf{p}}_k, \mathbf{w}_k)$   $\triangleright$  See Algorithm 3.27
18:     $L_{\text{votes}} \leftarrow (L_{\text{votes}}, \hat{\mathbf{p}}_k)$ 
19:     $L_{\text{decodedVotes}} \leftarrow (L_{\text{decodedVotes}}, \hat{\mathbf{v}}_k)$ 
20:     $L_{\text{writeIns}} \leftarrow (L_{\text{writeIns}}, \hat{\mathbf{s}}_k)$ 
21:     $k \leftarrow k + 1$ 
22:  end if
23: end for

```

Output:

List of ~~all selected encoded voting options~~ decrypted votes $L_{\text{votes}} = (\hat{\mathbf{p}}_0, \dots, \hat{\mathbf{p}}_{\hat{N}_c-1}) \in (((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^\psi)^{\hat{N}_c}$

List of ~~all selected decoded voting options~~ decoded votes $L_{\text{decodedVotes}} = (\hat{\mathbf{v}}_0, \dots, \hat{\mathbf{v}}_{\hat{N}_c-1}) \in ((\mathcal{T}_1^{50})^\psi)^{\hat{N}_c}$

List of ~~all selected decoded write-in votes~~ decoded write-ins $L_{\text{writeIns}} = (\hat{\mathbf{s}}_0, \dots, \hat{\mathbf{s}}_{\hat{N}_c-1}) \in ((\mathbb{A}_{\text{latin}}^*)^*)^{\hat{N}_c}$ \triangleright The alphabet $\mathbb{A}_{\text{latin}}$ is defined in section 3.8.1

Test values for the algorithm 6.12 are provided in [process-plaintexts.json](#).

6.3.5 Creating the Tally Files

The tally control component tallies the decoded voting-option-votes and decoded write-ins from the algorithm 6.12 ProcessPlaintexts in three files one file:

- The evoting-decrypt XML aggregates the actual (decoded) voting options from all ballot boxes into a single file, in the format defined by evoting-decrypt-1-3.xsd,
- The eCH-0222 XML eCH-0222 XML standardizes the decrypted ballots, in the format defined under ;
- The eCH-0110 XML indicates how many votes each voting option received, in the format defined under ; <http://www.ech.ch/xmlns/eCH-0222/3/eCH-0222-3-0.xsd>,

We assume a function CreateECH0222 that takes as input the election event configuration configuration XML, the key-value map of $L_{\text{decodedVotes}}$ per authorization name Map_{decodedVotes}, and the key-value map of L_{writeIns} per authorization name Map_{writeIns}, and outputs a eCH-0222 XML compliant with the eCH-0222 standard.

In particular, the tally control component's eCH-0110 XML eCH-0222 XML must declare votes for party lists without a candidate selection as invalid votes (if the election imposes this rule) as outlined in section 3.5.4.

The details of the generation of the tally files are eCH-0222 XML is beyond the scope of this document, but the schemas themselves are attached; we only outline its high-level structure. The eCH standard is a Swiss e-government standard that defines structured data formats to ensure interoperability in electronic communication between authorities and organizations. One of these standards, eCH-0222, defines the data types and event messages for exchanging raw data from electronic ballot boxes to the downstream systems responsible for processing and determining the results of an election event.

The eCH-0222 XML is primarily composed of three sections: the delivery header, the reporting body, and the raw data. The eCH-0222 XML combines the reporting body and the raw data into a raw data delivery. While the delivery header and reporting body primarily contain static information, the raw data section contains the structured, decoded votes of an election event.

Figure 16 illustrates the high-level structure of the eCH-0222 XML.

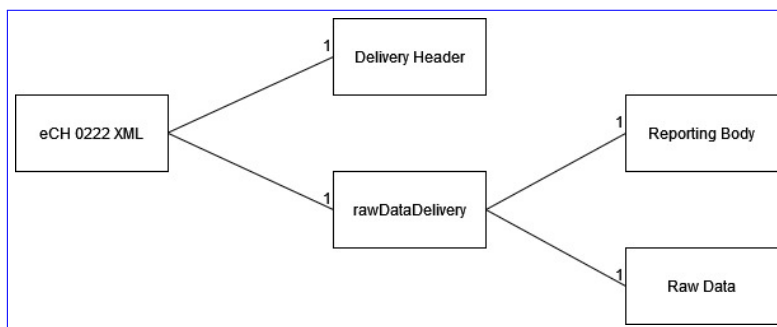


Fig. 16: High-level structure of the eCH-0222 XML.

6.3.6 Requesting a Proof of Non-Participation

The Federal Chancellery's Ordinance requires a proof of non-participation after the election period ended [7].

[VEleS art. 5 para. 2(b)]: A voter who has not cast his or her vote electronically can request proof after the electronic voting system is closed and within the statutory appeal deadlines that the trustworthy part of the system has not registered any vote cast using the client-side authentication credential of the voter.

The explanatory report clarifies that a procedural proof is sufficient to fulfill the requirement.

[Explanatory Report, Sec 4.2.1]: Regarding '...the attacker has not maliciously cast a vote on the voter's behalf which has subsequently been registered as a vote cast in conformity with the system and counted': such a proof would be of limited use during the ballot, as the attacker would still have time to cast a vote. Therefore, it is sufficient if voters can request this proof after the ballot. For reasons of efficiency, it is sufficient for the competent cantonal office to confirm to the voter that no vote has been cast on their behalf. The assumptions of trustworthiness set out in Number 2.9.1 apply to the examination by the competent body, and the auditors' technical aids may also be considered trustworthy. Furthermore, the requirement breaks the trust model, in that the attacker under Number 2.8 must not be able to access the client-side authentication credentials at all. With regard to the present requirement, the assumption must be made that the attacker has access to the client-side authentication credentials of individual voters.

After the tally phase, the voter can request the competent cantonal office to confirm that no vote has been cast on her behalf. The algorithm **RequestProofNonParticipation** specifies the procedure by which the competent cantonal office checks whether a confirmed vote corresponds to a specific voting card. Importantly, the algorithm **RequestProofNonParticipation** relies on a list of confirmed voting cards whose consistency and correctness was validated by the verifier. The voter expects the algorithm **RequestProofNonParticipation** to return \perp if she did not participate in the election event.

Algorithm 6.13 RequestProofNonParticipation

Context:

List of confirmed voting cards $\mathbf{vc}_1 \triangleright$ Stemming from the first control component that was previously validated by the verifier. The list contains the confirmed voting cards from all ballot boxes of the election event.

Verification card ID $\mathbf{vc}_{id} \in (\mathbb{A}_{Base16})^{1_{ID}}$

Operation:

```
1: if  $\mathbf{vc}_{id} \in \mathbf{vc}_1$  then
2:   participated =  $\top$ 
3: else
4:   participated =  $\perp$ 
5: end if
```

Output:

participated $\in \{\top, \perp\}$: \top if the voter successfully cast a vote, \perp otherwise.

7 Channel Security

7.1 Digital Signatures for Protocol Messages

The cryptographic protocol ensures the confidentiality of messages using encryption: the adversary cannot learn private information (such as the selected voting options) without controlling trustworthy components. Beyond confidentiality, the computational proof [20] details the requirements on the messages' authenticity in the section **Channel Security**.

Digital signatures ensure that the adversary cannot undetectably modify ciphertexts.

Furthermore, the sender signs additional information such as the election event ID to prevent replay attacks and allow the receiver to verify the context's correctness. Ultimately, the trustworthy protocol participants aim to achieve *injective agreement* on a protocol run's core messages.

The Swiss Post Voting System uses the signature scheme and key length specified in the crypto primitives specification. Each trustworthy party generates a signature key pair and distributes the corresponding certificate to the other trustworthy parties and the auditors. Each certificate is verified and then imported into the trustworthy party's trust store. All messages exchanged between the trustworthy parties are signed and then verified to be valid and to emanate from the expected party.

The generation of key pairs and corresponding certificates and the import of the certificates are regulated by specific processes, using the algorithm **GenKeysAndCert** from the crypto primitives specification for the generation and a manual process for the verification before the import.

The tables below identify the messages that are signed during each of the protocol phases. For each message, the signing party is identified, as well as the data that must be signed, along with any additional information required to prevent replay attacks. For each message, the signer calls the ~~GenSignature~~ GenSignature method defined in the crypto primitives specification, with the parameters given in the corresponding columns.

Each of the recipients listed must verify the validity of the signature with the method VerifySignature defined in the crypto primitives specification, using the certificate previously linked to the expected signing party and duly validated before being imported into the verifying party's trust store. As a side note, since there are no direct communication channels between the control components and the auditors, ~~the setup component forwards the~~ messages signed by the control components are forwarded to the auditors ~~; who by either the setup component or the tally control component.~~ The auditors then verify the ~~signature~~ signatures independently.

Usually, we sign data elements—and not the actual files into which the data elements are serialized. However, we deviate from this rule for XML files that serve as an important interface to upstream and downstream systems. Due to the complexity of these XML files' structure and the existence of widely recognized standards for XML signing, we use two specific algorithms for XML signatures, as outlined in section 7.2.

Table 15 highlights the messages that the parties exchange during the configuration phase.

Message Name	Signer	Recipient(s)	Message Content	Context Data
ElectionEventContext	Setup Comp.	Online CC _j , Tally CC, Auditors, Voting Server	$(p, q, g, \text{seed}, p, n_{\max}, \psi_{\max})$ $\delta_{\max}, \{vcs, pTable, N_E\}^{V_{vcs}}$ ElectionEventContextPayload , see table 18	("election event context", ee)
CantonConfig	Canton	Setup Comp., Auditors, Tally CC	configuration XML configuration XML ¹	("configuration") ~
ControlComponentPublicKeys	Online CC _j	Setup Comp., Auditors	$(pk_{CCR,j}, \pi_{pkCCR,j}, EL_{pk,j}, \pi_{ELpk,j})$ ControlComponentPublicKeysPayload , see table 18	("OnlineCC keys", j, ee)
SetupComponentVerification-Data	Setup Comp.	Online CC _j , Auditors	$(\{vc_{id}, K_{id}, c_{pCC,id}, c_{CK,id}\}_{id=0}^{N_E-1})$ SetupComponentVerificationDataPayload , see table 18	("verification data", ee, yes)
ControlComponentCodeShares	Online CC _j	Setup Comp., Auditors	$(\{vc_{id}, K_{j,id}, Kc_{j,id}, c_{expPCC,j,id}, c_{expCK,j,id}, \pi_{expPCC,j,id}, \pi_{expCK,j,id}\}_{id=0}^{N_E-1})$ ControlComponentCodeSharesPayload , see table 18	("encrypted code shares", j, ee, vcs)
SetupComponentLVCCAl- lowList	Setup Comp.	Online CC _j	L _{lvcc}	("lvcc allow list", ee, vcs)
SetupComponentEvotingPrint	Setup Comp.	Printing Comp.	evoting print XML evoting print XML ¹	("evoting print") ~
SetupComponentCMTable	Setup Comp.	Voting Server	CMtable ²	("cm table", ee, vcs)
SetupComponentVerification- CardKeyStores	Setup Comp.	Voting Server	VCKs	("vc keystore", ee, vcs)
SetupComponentVoterAuthen- ticationData	Setup Comp.	Voting Server	(credentialID, hAuth)	("voter authentication", ee, vcs)
SetupComponentPublicKeys	Setup Comp.	Online CC _j , Tally CC, Auditors, Voting Server	$(j, pk_{CCR,j}, \pi_{pkCCR,j}, EL_{pk,j}, \pi_{ELpk,j})$ SetupComponentPublicKeysPayload , see table 18	("public keys", "setup", ee)
SetupComponentTallyData	Setup Comp.	Auditors, Tally CC	(ve, K) SetupComponentTallyDataPayload , see table 18	("tally data", ee, vcs)
SetupComponentElectoral- BoardHashes	Setup Comp.	Tally CC	(hPW ₀ , ..., hPW _{k-1})	("electoral board hashes", ee)

Tab. 15: Overview of the authenticated messages and their signers and recipients in the configuration phase as displayed in figures 7 and 8.

¹ [This message is signed using the XML signature algorithm outlined in section 7.2.](#)

² The CMtable is ordered as per algorithm 4.9, and represented as a vector of key, value pairs, *i.e.* $((k_0, v_0), \dots, (k_{N_E \cdot (n+1) - 1}, v_{N_E \cdot (n+1) - 1}))$.

The voting phase provides implicit authentication: the voting client can only generate extractable short Choice Return Codes using the correct verification card secret key k_{id} , and the control components only derive the short Vote Cast Return Code VCC_{id} from a valid Ballot Casting Key BCK_{id} .

However, the elements exchanged between the control components and the voting server should still be authenticated, as shown in table 16. Signing the voting server's messages is not necessary from the trust model's point of view since the voting server is an *untrusted* component. Nevertheless, it provides an additional defense-in-depth mechanism against attackers controlling the network between the voting server and the control components.

Message Name	Signer	Recipient(s)	Message Content	Context Data
VotingServerEncryptedVote	Voting server	Online CC_j	$(E1, E2, \tilde{E1}, \pi_{Exp}, \pi_{EqEnc})$	("encrypted vote", ee, vcs, vc_{id})
ControlComponentPartialDecrypt	Online CC_j	Online $CC_{j'}$	$(d_j, \pi_{decPCC,j})$	("partial decrypt", j, ee, vcs, vc_{id})
ControlComponentLCCShare <u>ControlComponentlCCShare</u>	Online CC_j	Voting Server	$lCC_{j,id}$ <u>ControlComponentlCCSharePayload,</u> <u>see table 18</u>	("lcc share", j, ee, vcs, vc_{id})
VotingServerConfirm	Voting server	Online CC_j	CK_{id}	("confirmation key", ee, vcs, vc_{id})
ControlComponenthlVCC <u>ControlComponenthlVCCShare</u>	Online CC_j	Online $CC_{j'}$	$hlVCC_{j,id}$ <u>ControlComponenthlVCCSharePayload,</u> <u>see table 18</u>	("hlvcc", j, ee, vcs, vc_{id})
ControlComponentlVCCShare	Online CC_j	Voting Server	$lVCC_{j,id}$ <u>ControlComponentlVCCSharePayload,</u> <u>see table 18</u>	("lvcc share", j, ee, vcs, vc_{id})

Tab. 16: Overview of the authenticated messages and their signers and recipients in the voting phase as displayed in figures 11 and 12.

Table 17 summarizes the messages of the tally phase.

Message Name	Signer	Recipient(s)	Message Content	Context Data
ControlComponentVotesHash	Online CC _j	Online CC _{j'}	hvc _j	(“voteshash”, j, ee, bb)
ControlComponentBallotBox	Online CC _j	Tally CC, Auditors	$(\{vc_{j,1}, E1_{j,1}, \tilde{E1}_{j,1}, E2_{j,1}, \pi_{exp,j,1}, \pi_{eqbnc,j,1}\}_{i=0}^{N_E-1})$ ControlComponentBallotBoxPayload, see table 18	(“ballotbox”, j, ee, bb)
ControlComponentShuffle	Online CC _j	Online CC _{j'} , Tally CC, Auditors	$(c_{mix,j}, \pi_{mix,j}, c_{dec,j}, \pi_{dec,j})$ ControlComponentShufflePayload, see table 18	(“shuffle”, j, ee, bb)
TallyComponentShuffle	Tally CC	Auditors	$(c_{mix,5}, \pi_{mix,5}, m, \pi_{dec,5})$ TallyComponentShufflePayload, see table 18	(“shuffle”, “offline”, ee, bb)
TallyComponentVotes	Tally CC	Auditors	$(L_{votes}, L_{decodedVotes}, L_{writeIns})$ TallyComponentVotesPayload, see table 18	(“decoded votes”, ee, bb)
TallyComponentDecrypt <u>TallyComponentEch0222</u>	Tally CC	Auditors	evoting-decrypt-XML <u>eCH-0222 XML³</u>	(“evoting-decrypt”) <u>(“eCH-0222”)</u>
TallyComponentEch0222 <u>ControlComponentExtractedElectionEvent</u>	Online CC_j <u>Online CC_j</u>	Auditors <u>Dispute Resolver</u>	eCH-0222-XML <u>ControlComponentExtractedElectionEventPayload</u> see table 18	(“eCH-0222”) <u>(“ControlComponentExtractedElectionEvent”, j, ee)</u>
TallyComponentEch0110 <u>ControlComponentExtractedVerificationCards</u>	Online CC_j <u>Online CC_j</u>	Auditors <u>Dispute Resolver</u>	eCH-0110-XML <u>ControlComponentExtractedVerificationCardsPayload</u> see table 18	(“eCH-0110”) <u>(“ControlComponentExtractedVerificationCards”, j, ee)</u>
<u>DisputeResolverResolvedConfirmedVotes</u>	<u>Dispute Resolver</u>	<u>Online CC_j</u>	<u>DisputeResolverResolvedConfirmedVotesPayload</u> see table 18	<u>(“DisputeResolverResolvedConfirmedVotes”, ee)</u>

Tab. 17: Overview of the authenticated messages and their signers and recipients in the tally phase as displayed in figure 13.

~~The XML files are signed according to the following high-level process~~

We refer to the combination of the message content and its digital signature as the *payload* and we accompany each payload with a JSON schema. A JSON schema is a JSON document that defines the structure, constraints, and data types of JSON data, used to describe the message content and its alignment with the system specification. Table 18 provides an overview of the schemas of the JSON payloads.

³ This message is signed using the XML signature algorithm outlined in section 7.2.

Payload Name	Referenced schemas
<u>ControlComponentBallotBoxPayload</u>	<u>GqGroup, EncryptedVerifiableVote, ContextIds, ElGamalMultiRecipientCiphertext, ExponentiationProof, PlaintextEqualityProof, CryptoPrimitivesSignature</u>
<u>ControlComponentCodeSharesPayload</u>	<u>GqGroup, ControlComponentCodeShare, ElGamalMultiRecipientCiphertext, ExponentiationProof, CryptoPrimitivesSignature</u>
<u>ControlComponentExtractedElectionEventPayload</u>	<u>ExtractedElectionEvent, GqGroup, ExtractedVerificationCardSet, CryptoPrimitivesSignature</u>
<u>ControlComponentExtractedVerificationCardsPayload</u>	<u>GqGroup, ExtractedVerificationCard, CryptoPrimitivesSignature</u>
<u>ControlComponenth1VCCSharePayload</u>	<u>GqGroup, ConfirmationKey, CryptoPrimitivesSignature</u>
<u>ControlComponentl1CCSharePayload</u>	<u>GqGroup, LongChoiceReturnCodeShare, CryptoPrimitivesSignature</u>
<u>ControlComponentlVCCSharePayload</u>	<u>GqGroup, LongVoteCastReturnCodeShare, ConfirmationKey, CryptoPrimitivesSignature</u>
<u>ControlComponentPublicKeysPayload</u>	<u>GqGroup, ControlComponentPublicKeys, SchnorrProof, CryptoPrimitivesSignature</u>
<u>ControlComponentShufflePayload</u>	<u>GqGroup, VerifiableShuffle, ElGamalMultiRecipientCiphertext, ShuffleArgument, ProductArgument, HadamardArgument, ZeroArgument, SingleValueProductArgument, MultiExponentiationArgument, VerifiableDecryptions, CryptoPrimitivesSignature</u>
<u>DisputeResolverResolvedConfirmedVotesPayload</u>	<u>CryptoPrimitivesSignature</u>
<u>ElectionEventContextPayload</u>	<u>GqGroup, ElectionEventContext, VerificationCardSetContext, PrimesMappingTable, PrimesMappingTableEntry, VoteTexts, Ballot, ElectionTexts, ElectionInformation, Election, Candidate, List, EmptyList, WriteInPosition, CryptoPrimitivesSignature</u>
<u>SetupComponentPublicKeysPayload</u>	<u>GqGroup, SetupComponentPublicKeys, ControlComponentPublicKeys, SchnorrProof, CryptoPrimitivesSignature</u>
<u>SetupComponentTallyDataPayload</u>	<u>GqGroup, CryptoPrimitivesSignature</u>
<u>SetupComponentVerificationDataPayload</u>	<u>GqGroup, SetupComponentVerificationData, ElGamalMultiRecipientCiphertext, CryptoPrimitivesSignature</u>
<u>TallyComponentShufflePayload</u>	<u>GqGroup, VerifiableShuffle, ElGamalMultiRecipientCiphertext, ShuffleArgument, ProductArgument, HadamardArgument, ZeroArgument, SingleValueProductArgument, MultiExponentiationArgument, VerifiablePlaintextDecryption, CryptoPrimitivesSignature</u>
<u>TallyComponentVotesPayload</u>	<u>GqGroup, CryptoPrimitivesSignature</u>

Tab. 18: Overview of the JSON schemas of the payloads exchanged.

The other JSON schema files in alphabetical order are ConfirmationKey, ControlComponentPublicKeys, CryptoPrimitivesSignature, ElGamalMultiRecipientCiphertext, ExponentiationProof, GqGroup, HadamardArgument, MultiExponentiationArgument, ProductArgument, SchnorrProof, SingleValueProductArgument, ShuffleArgument, VerifiableShuffle, ZeroArgument.

7.2 Digital Signatures for File Interfaces (XML)

The Swiss Post Voting System communicates with upstream and downstream systems using XML files. Given the complexity of these XML structures, providing detailed pseudo-code describing their structure would be impractical. Instead, we rely on the standardized XML signature syntax and verification methods as defined in [1], which are supported by most programming languages.

Within the Swiss Post Voting System, we serialize the following messages as XML files:

- The message `CantonConfig` serialized in the file `configuration XML` as indicated in table 15.
- The message `SetupComponentEvotingPrint` serialized in the file `evoting-print XML` as indicated in table 15.
- The message `TallyComponentEch0222` serialized in the file `eCH-0222 XML` as indicated in table 17.

Accordingly, we define Algorithm 7.1 for signing XML files and Algorithm 7.2 for verifying XML signatures.

We assume that the recipient validates the XML messages against the corresponding schema definition (XSD) to ensure that the XML adheres to the expected structure and data constraints.

Subsequently, the signature generation and verification method follows the W3C XML signature standard [1]. Signing an XML file follows a structured series of steps:

1. `CreateSignedInfo`: The `<SignedInfo>` element is instantiated with the following parameters:
 - ~~starting from the root element of the XML file,~~ `Reference URI`: An empty string (`""`) to indicate that the entire document is signed.
 - ~~each `complexType` element is represented as a nested vector of values within the domain accepted by `RecursiveHash`,~~ `CanonicalizationMethod`: The exclusive C14N canonicalization algorithm[4].
 - ~~within such `complexType`, elements are taken in the order in which they are defined in~~ `Transforms`: the enveloped-signature transform.
 - `DigestMethod`: The digest algorithm. We use the SHA-256 algorithm.
 - `SignatureMethod`: The signature algorithm. We use the RSAPSS algorithm with 3072-bits key size.
2. `Canonicalize`: Prior to any cryptographic operation, the XML file is *canonicalized*. This step removes ambiguities such as OS-dependent line breaks, varying whitespace, and attribute ordering differences, ensuring that any semantically equivalent XML produces the same canonical form.
3. `ApplyTransforms`: The data referenced by the signature undergoes the enveloped-signature transform, which removes the signature element itself from the document prior to digest calculation.

4. Digest: The transformed data is hashed using the SHA-256 digest algorithm.
5. Update: Update the `<SignedInfo>` with the digest.
6. Sign: The canonicalized `<SignedInfo>` element is signed using the Channel Security keystore described in section 7 with the RSAPSS algorithm. This signature is stored in the `<SignatureValue>` element.
7. EmbedSignature: The computed signature is embedded into the corresponding XML file. For the XML files in the Swiss Post Voting System, we embed the signature as follows:
 - CantonConfig (configuration XML): Embed the `<Signature>` element as the last child of the `XSD` root element.
 - ~~if an element is optional and absent~~, SetupComponentEvotingPrint (evoting-print XML): Embed the `<Signature>` element as the last child of the root element.
 - TallyComponentEch0222 (eCH-0222 XML): Embed the `<Signature>` element in the ~~string ``no-<tokenname>-value`` is hashed with~~ `rawData/extensions` element.

We omit embedding the certificate into the `<SignedInfo>` element since we assume that the recipient knows the sender's public key and ensured its authenticity—as outlined in section 7.1. Following these steps, we instantiate the algorithm `GenXMLSignature` as follows.

Algorithm 7.1 GenXMLSignature

Input:

XML document D
Secret key of the signer sk

Operation: ▷ The algorithm follows the XML signature standard[1] and uses the parameters defined above.

- 1: $SI \leftarrow \text{CreateSignedInfo}(D)$
 - 2: $SI_{\text{can}} \leftarrow \text{Canonicalize}(SI)$
 - 3: $T \leftarrow \text{ApplyTransforms}(D)$
 - 4: $d \leftarrow \text{Digest}(T, \text{digestAlg})$
 - 5: $SI \leftarrow \text{Update}(SI, d)$
 - 6: $SV \leftarrow \text{Sign}(SI_{\text{can}}, sk, \text{sigAlg})$
 - 7: $D_{\text{signed}} \leftarrow \text{EmbedSignature}(D, SI, SV)$
-

Output:

Signed XML document D_{signed}

Upon receipt, the signature is verified by applying the same canonicalization and transformation procedures to the XML document, recalculating the digest, and using the public key to validate the <SignatureValue>.
The verification algorithm proceeds as follows:

1. ExtractSignedInfo: Extract the <SignedInfo> element from the signed XML document. Depending on the XML file type, the location of the corresponding <Signature> element varies:
 - CantonConfig (configuration XML): Extract the <Signature> element from the last child of the root element.
 - SetupComponentEvotingPrint (evoting-print XML): Extract the <Signature> element from the last child of the root element.
 - TallyComponentEch0222 (eCH-0222 XML): Extract the <Signature> element from the rawData/extensions element.
2. Canonicalize: Prior to any cryptographic operation, the XML file is canonicalized. This step removes ambiguities such as OS-dependent line breaks, varying whitespace, and attribute ordering differences, ensuring that any semantically equivalent XML produces the same canonical form. We use the exclusive C14N canonicalization algorithm[4].
3. ApplyTransforms: The data referenced by the signature undergoes the enveloped-signature transform, which removes the signature element itself from the document prior to digest calculation.
4. Digest: The transformed data is hashed using the SHA-256 digest algorithm.
5. ExtractDigest: Retrieve the digest value from the <DigestValue> element contained within the <SignedInfo> section.
6. Check the digest: Compare the digest computed from the transformed data with the extracted digest value. Return false if the digest values do not match.
7. ExtractSignatureValue: Extract the signature value from the <SignatureValue> element of the ~~value of tokenname being replaced with the token name, to prevent trivial collisions in case of several optional elements following each other, signed XML document.~~
8. ~~each simpleType is converted as follows: number-like elements take their number representation, boolean values are converted into their canonical string representation ('true' or 'false'), binary values are represented as byte-arrays and string-like values (e.g. normalizedString, token, .~~ Verify: Use the public key and the specified signature algorithm to verify that the signature is valid for the canonicalized <SignedInfo> element. If the signature is valid, the verification returns true; otherwise, it returns false.

Algorithm 7.2 VerifyXMLSignature

Input:

Signed XML document D_{signed}
Public key of the signer pk

Operation: ▷ The algorithm follows the XML signature standard[1] and uses the parameters defined above.

```
1:  $SI \leftarrow \text{ExtractSignedInfo}(D_{\text{signed}})$ 
2:  $SI_{\text{can}} \leftarrow \text{Canonicalize}(SI)$ 
3:  $T \leftarrow \text{ApplyTransforms}(D_{\text{signed}})$ 
4:  $d' \leftarrow \text{Digest}(T, \text{digestAlg})$ 
5:  $d \leftarrow \text{ExtractDigest}(SI)$ 
6: if  $d' \neq d$  then
7:   return  $\perp$ 
8: end if
9:  $SV \leftarrow \text{ExtractSignatureValue}(D_{\text{signed}})$ 
10: return  $\text{Verify}(SI_{\text{can}}, SV, pk, \text{sigAlg})$ 
```

Output:

\top if the signature is valid, \perp otherwise.

7.3 Streamable Symmetric Encryption and Decryption for Dataset Security

Whenever a dataset is exported for transfer to or from an offline component, the dataset is symmetrically encrypted as an additional security mechanism. A dataset contains cryptographic protocol messages serialized in a standard file format such as JSON (JavaScript Object Notation) or XML (Extensible Markup Language). The serialization of the messages is outside the scope of this document.

We use an authenticated symmetric encryption scheme based on Authenticated Encryption with Associated Data (AEAD) as defined in the crypto primitives specification. AEAD protects the dataset in transit from eavesdropping and tampering by ensuring both the confidentiality and integrity of its content.

We require an encryption and decryption mechanism that can handle datasets too large to fit entirely in memory. To achieve this, we process these datasets using a streaming approach. Streaming, in this context, means breaking the data into smaller chunks and processing each chunk sequentially, rather than loading the entire dataset into memory at once.

The files of a dataset are collected as a non-compressed ZIP folder using the DEFLATE method according to [5, RFC 1951]. This process produces a single file with a deterministic hash value, enabling auditors to verify and record the dataset. Moreover, we assume the availability of methods for zipping and unzipping a dataset, along with a unique byte-array representation of the resulting ZIP file, enabling the transformation of the dataset into a byte array. Algorithms 7.3 and 7.4 specify the streamable encryption and decryption of a byte array.

By default, we instantiate the algorithms in this section with an empty byte array as associated data `associated`.

We assume the existence of functions `readNextBlock` and `endOfData` to process streams of arbitrary byte arrays. For the definitions of the `AuthenticatedEncryption` and `AuthenticatedDecryption` functions, we refer the reader to the section Symmetric Authenticated Encryption in the crypto primitives specification.

As explained in the crypto-primitives specification, it is critical that nonces should not be reused. Algorithm 7.3 should not be used in a virtualized environment, as a rollback could lead to a nonce reuse.

Algorithm 7.3 GenStreamCiphertext

Input:

The plaintext $P \in \mathcal{B}^*$

A password $\in \mathbb{A}_{UCS}^*$

Associated data $associated \in \mathcal{B}^*$ \triangleright By default, we instantiate the algorithm with an empty byte array as associated data.

Operation:

- 1: $(derivedKey, salt) \leftarrow \text{GenArgon2id}(\text{toBytes}(\text{password}))$ \triangleright see crypto primitives specification
 - 2: $nonce \leftarrow \text{RandomBytes}(12)$ \triangleright see crypto primitives specification
 - 3: $C \leftarrow \langle \rangle$
 - 4: **while** $\neg \text{endOfData}(P)$ **do**
 - 5: $block \leftarrow \text{readNextBlock}(P)$
 - 6: $encryptedBlock \leftarrow \text{AuthenticatedEncryption}(derivedKey, nonce, block, associated)$ \triangleright see crypto primitives specification
 - 7: $C \leftarrow C \parallel encryptedBlock$
 - 8: **end while**
-

Output:

The salt, nonce and authenticated ciphertext $salt \parallel nonce \parallel C \in \mathcal{B}^*$

\triangleright $salt \in \mathcal{B}^{16}, nonce \in \mathcal{B}^{12}$

Test values for the algorithm 7.3 are provided in [gen-stream-ciphertext.json](#).

Algorithm 7.4 GetStreamPlaintext

Input:

The salt, nonce and authenticated ciphertext $\text{salt} \parallel \text{nonce} \parallel C \in \mathcal{B}^*$
 $\triangleright \text{salt} \in \mathcal{B}^{16}, \text{nonce} \in \mathcal{B}^{12}$
The password $\text{password} \in \mathbb{A}_{UCS}^*$
Associated data $\text{associated} \in \mathcal{B}^*$ \triangleright By default, we instantiate the algorithm with an empty
byte array as associated data.

Operation:

```
1:  $\text{derivedKey} \leftarrow \text{GetArgon2id}(\text{toBytes}(\text{password}), \text{salt})$   $\triangleright$  see crypto primitives specification
2:  $P \leftarrow \langle \rangle$ 
3: while  $\neg \text{endOfData}(C)$  do
4:    $\text{block} \leftarrow \text{readNextBlock}(C)$ 
5:    $\text{decryptedBlock} \leftarrow \text{AuthenticatedDecryption}(\text{derivedKey}, \text{nonce}, \text{associated}, \text{block})$ 
 $\triangleright$  see crypto primitives specification
6:    $P \leftarrow P \parallel \text{decryptedBlock}$ 
7: end while
```

Output:

The plaintext $P \in \mathcal{B}^*$
Test values for the algorithm 7.4 are provided in get-stream-plaintext.json. ~~) and also date-like
types are represented as strings, the signature field itself must be ignored.~~

Acknowledgements

Swiss Post is thankful to all security researchers for their contributions and the opportunity to improve the system's security guarantees. In particular, we want to thank the following experts for their reviews or suggestions reported on our [Gitlab repository](#). We list them here in alphabetical order:

- Veronique Cortier, Pierrick Gaudry, Alexander Debant (Université de Lorraine, CNRS, Inria)
- Aleksander Essex (Western University Canada)
- Rolf Haenni, Reto Koenig, Philipp Locher, Eric Dubuis (Bern University of Applied Sciences)
- Thomas Edmund Haines (Australian National University)
- Olivier Pereira (Universtité catholique Louvain)
- Ruben Santamarta (reversemode)
- Carsten Schürmann (IT University of Copenhagen)
- Vanessa Teague (Thinking Cybersecurity)
- Marc Wyss (ETHZ)

References

- [1] M. Bartel et al.: *XML Signature Syntax and Processing Version 1.1*. Ed. by D. Eastlake et al. <http://www.w3.org/TR/2013/REC-xmlsig-core1-20130411/>. W3C Recommendation, 11 April 2013.
- [2] D. Bernhard et al.: “Verifiability analysis of CHVote”. In: *Cryptology ePrint Archive* (2018).
- [3] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson: *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC 9106. 2021. DOI: 10.17487/RFC9106. URL: <https://www.rfc-editor.org/info/rfc9106>.
- [4] J. Boyer and G. Marcy: *Canonical XML Version 1.1*. <http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/>. W3C Recommendation, 2 May 2008.
- [5] P. Deutsch: *DEFLATE Compressed Data Format Specification*. RFC 1951. 1996. DOI: 10.17487/RFC1951. URL: <https://www.rfc-editor.org/info/rfc1951>.
- [6] Die Schweizerische Bundeskanzlei (BK): *E-voting Catalogue of Measures by the Confederation and Cantons, Approved by the Vote électronique Steering Committee (SC VE): 4 August 2023*. 2023. URL: https://www.bk.admin.ch/dam/bk/en/dokumente/pore/E_Voting/E-voting%20Catalogue%20of%20measures%20by%20the%20Confederation%20and%20cantons,%204%20August%202023.pdf.download.pdf.
- [7] Die Schweizerische Bundeskanzlei (BK): *Federal Chancellery Ordinance on Electronic Voting (OEV), 01 July 2022*.
- [8] Die Schweizerische Bundeskanzlei (BK): *Partial revision of the Ordinance on Political Rights and total revision of the Federal Chancellery Ordinance on Electronic Voting (Redesign of Trials). Explanatory report for its entry into force on 1 July 2022*.
- [9] eCH Association: *eCH-0045 Datenstandard Stimm- und Wahlregister V4.1.0*. eCH E-Government Standards. Version 4.1.0. Approved on June 2, 2022, published on April 19, 2023. 2023. URL: <https://www.ech.ch/de/ech/ech-0045/4.1.0>.
- [10] eCH Association: *eCH-0155 Datenstandard politische Rechte V4.1*. eCH E-Government Standards. Version 4.1. 2021. URL: <https://www.ech.ch/de/ech/ech-0155/4.1>.
- [11] Eidgenössisches Departement für auswärtige Angelegenheiten EDA: *Swiss Political System - Direct Democracy*. <https://www.eda.admin.ch/aboutswitzerland/en/home/politik/uebersicht/direkte-demokratie.html/>. Retrieved on 2020-07-15.
- [12] gfs.bern: *Vorsichtige Offenheit im Bereich digitale Partizipation - Schlussbericht*. 2020.
- [13] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis: *CHVote Protocol Specification, Version 3.5*. Cryptology ePrint Archive, Report 2017/325. <https://eprint.iacr.org/2017/325>. 2023.
- [14] T. Haines, S. J. Lewis, O. Pereira, and V. Teague: *How not to prove your election outcome*. 2020.
- [15] S. Josefsson et al.: *The base16, base32, and base64 data encodings*. Tech. rep. RFC 4648, October, 2006.

- [16] D. M'Raihi, J. Rydell, M. Pei, and S. Machani: *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. 2011. DOI: 10.17487/RFC6238. URL: <https://www.rfc-editor.org/info/rfc6238>.
- [17] C. Percival: *Stronger key derivation via sequential memory-hard functions*. 2009.
- [18] B. Smyth: "A foundation for secret, verifiable elections." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 225.
- [19] Swiss Post: *Cryptographic Primitives of the Swiss Post Voting System. Pseudocode Specification. Version 1.5.0*. <https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives>. 2025.
- [20] Swiss Post: *Protocol of the Swiss Post Voting System. Computational Proof of Complete Verifiability and Privacy. Version 1.4.0*. <https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Protocol>. 2025.
- [21] Swiss Post: *Swiss Post Voting System. Verifier Specification. Version 1.6.0*. <https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/System>. 2025.

List of Algorithms

3.1	GetElectionEventEncryptionParameters	23
3.2	GetEncodedVotingOptions	32
3.3	GetActualVotingOptions	33
3.4	GetSemanticInformation	33
3.5	GetCorrectnessInformation	34
3.6	GetBlankCorrectnessInformation	35
3.7	GetWriteInEncodedVotingOptions	36
3.8	GetPsi	37
3.9	GetDelta	37
3.10	Factorize	41
3.11	<u>GetHashElectionEventContext</u>	43
3.12	<u>GetHashContext</u>	44
3.13	ExtractElectionEvent	46
3.14	ExtractVerificationCardSet	46
3.15	<u>GetHashExtractedElectionEvent</u>	47
3.16	ExtractVerificationCards	48
3.17	<u>VerifyKeyGenerationSchnorrProofs</u>	50
3.18	<u>VerifyCCSchnorrProofs</u>	51
3.19	DeriveCredentialId	52
3.20	DeriveBaseAuthenticationChallenge	53
3.21	WriteInToQuadraticResidue	55
3.22	WriteInToInteger	56
3.23	QuadraticResidueToWriteIn	57
3.24	IntegerToWriteIn	57
3.25	EncodeWriteIns	58
3.26	IsWriteInOption	59
3.27	DecodeWriteIns	60
4.1	GenSetupData	65
4.2	GenVerDat	66
4.3	GetVoterAuthenticationData	67
4.4	GenKeysCCR	68
4.5	GenEncLongCodeShares	70
4.6	CombineEncLongCodeShares	72
4.7	<u>VerifyEncryptedPCCExponentiationProofs</u>	73
4.8	<u>VerifyEncryptedCKExponentiationProofs</u>	74
4.9	GenCMTable	75
4.10	GenVerCardSetKeys	76
4.11	GenCredDat	77
4.12	SetupTallyCCM	79
4.13	SetupTallyEB	81
5.1	GetAuthenticationChallenge	87
5.2	VerifyAuthenticationChallenge	89
5.3	GetKey	91
5.4	CreateVote	95
5.5	VerifyBallotCCR	96

5.6	PartialDecryptPCC	98
5.7	DecryptPCC	99
5.8	CreateLCCShare	101
5.9	ExtractCRC	102
5.10	CreateConfirmMessage	104
5.11	CreateLVCCShare	105
5.12	VerifyLVCCHash	106
5.13	ExtractVCC	107
6.1	GetMixnetInitialCiphertexts	113
6.2	VerifyMixDecOnline	114
6.3	MixDecOnline	116
6.4	CheckExtractedElectionEventConsistency	120
6.5	CheckVoteConsistency	121
6.6	ConfirmVoteAgreement -CheckVoteConfirmationConsistency	122
6.7	<u>ConfirmVoteAgreement</u>	123
6.8	<u>UpdateConfirmedVotingCards</u>	124
6.9	VerifyVotingClientProofs	125
6.10	VerifyMixDecOffline	126
6.11	MixDecOffline	127
6.12	ProcessPlaintexts	129
6.13	RequestProofNonParticipation	132
7.1	GenXMLSignature	140
7.2	VerifyXMLSignature	142
7.3	<u>GenStreamCiphertext</u>	144
7.4	<u>GetStreamPlaintext</u>	145

List of Figures

1	Example code sheet	10
2	Example voting instructions	11
3	Two-round return code scheme	12
4	Alternative voting process after sending vote	13
5	Alternative voting process after confirming the vote	13
6	Overview of the election event context	25
7	SetupVoting sequence diagram	63
8	SetupTally sequence diagram	78
9	Finalize configuration phase sequence diagram	82
10	AuthenticateVoter sequence diagram	86
11	SendVote sequence diagram	93
12	ConfirmVote sequence diagram	103
13	Tally phase sequence diagram	110
14	MixOnline sequence diagram	112
15	DisputeResolver sequence diagram	118
16	eCH0222 high-level structure	130

List of Tables

1	Overview of the Voter's codes	20
2	List of the Voting Client's keys	20
3	List of the Control Component's keys	21
4	List of the Setup Component's keys	21
5	List of the Tally Control Component's keys	22
6	Overview of the Voter's codes.	24
7	Overview of the algorithms' context	27
8	Example parameters of the electoral model.	29
9	Verification card set specific parameters	29
10	Election event specific parameters	29
11	Configuration phase algorithms overview	62
12	Voting phase algorithms overview	84
13	Context and input validations to authenticate voter	88
14	Tally phase algorithms overview	109
15	Overview of authenticated messages in the configuration phase	134
16	Overview of authenticated messages in the voting phase	135
17	Overview of authenticated messages in the tally phase	136
18	Overview of the JSON schemas of the payloads exchanged.	137