Swiss Post Voting System

Verifier Specification

Swiss Post*

Version 1.6.0

Abstract

The Swiss Post Voting System allows citizens to vote remotely in a secure and verifiable manner. Independent auditors can check that the system worked correctly. For that means, the system generates verifiable cryptographic evidence. The auditor reviews the provided evidence using a verifier software. The necessary verifications correspond to two algorithms: VerifyConfigPhase and VerifyTally. This document details the verifications that the verifier software has to implement. At the same time, it serves as a detailed specification of the Swiss Post verifier, which is an open-source software application to verify the Swiss Post Voting System. Therefore, this document serves as a manual for developing an independent verifier software and validating or extending the Swiss Post verifier.

Disclaimer

Revision chart

Version	Description	Author	Reviewer	Date
0.9	First published version	OE	XM, TH	2021-09-01
0.9.1		OE, HR	XM, JS, TH	2021-10-15
0.9.2		OE, HR	XM, JS, TH	2022-02-17
1.0.0	First full version	TH, OE	XM, JS, HR	2022-06-24
1.0.1	See change log for version 1.0.1	TH, OE	XM, JS, HR	2022-08-19
1.1.0	See change log for version 1.1.0	TH, OE	XM, JS, HR	2022-10-03
1.2.0	See change log for version 1.2.0	TH, OE	XM, JS, HR	2022-10-31
1.3.0	See change log for version 1.3.0	AH, OE	XM, JS, TH	2022-12-09
1.3.1	See change log for version 1.3.1	AH, OE	XM, JS, TH	2023-02-23
1.4.0	See change log for version 1.4.0	AH, OE	XM, JS	2023-04-19
1.4.1	See change log for version 1.4.1	AH, OE	XM, JS	2023-06-16
1.5.0	See change log for version 1.5.0	AH, CC, CK, OE	XM, JS	2024-02-14
1.5.1	See change log for version 1.5.1	AH, CC, CK, OE	XM, JS	2024-03-29
1.5.2	See change log for version 1.5.2	AH, CC, CK, OE	XM, JS	2024-06-17
1.6.0	See change log for version 1.6.0	AH, CC, CK, OE	XM, JS	2025-06-20

Contents

Symbols

$\mathbb{A}_{Base16}$	Base16 (Hex) alphabet [7]
$\mathbb{A}_{\mathsf{latin}}$	Extended Latin alphabet, as described in the crypto primitives specification
$\mathbb{A}_{UCS}$	Alphabet of the Universal Coded Character Set (UCS) according to ISO/IEC10646
$\mathcal{T}_1^{50}$	Alphabet used for voting option identifiers (word characters and minus sign: `[\w\-]{1,50}`)
$\mathcal{B}^*$	Set of byte arrays of arbitrary length
$\mathtt{bb}$	Ballot box ID
$\mathbf{bb}$	Vector of ballot box IDs of an election event
$\mathbf{c}_{\mathsf{Dec}}$	List of partially decrypted ciphertexts
$\mathtt{CK}$	Confirmation Key
CCM	Mixing control components
CCR	Return Codes control components
δ	Number of write-in options + 1 for a specific verification card set
$\delta_{\mathtt{max}}$	Maximum number of write-in options + 1 across all verification card sets
δ_{sup}	Maximum supported number of write-in options + 1
$\mathtt{E1}$	Encrypted vote
$\widetilde{\mathtt{E1}}$	Exponentiated encrypted vote
$\mathtt{E2}$	Encrypted partial Choice Return Codes
$\mathtt{ee}$	Election event id
g	Generator of the encryption group
$\mathbb{G}_q$	Set of quadratic residues modulo p of size q. The computational proof refers to this set as $\mathbb{Q}_p$
$\mathtt{l}_{\mathtt{ID}}$	Character length of unique identifiers
L_{votes}	List of decrypted votes
$L_{\mathsf{decodedVotes}}$	List of decoded votes
L_{writeIns}	List of decoded write-ins
$\mathbf{m}$	List of plaintext votes
$\mathbb{N}^+$	Set of strictly positive integer numbers
$\mathtt{N}_{\mathtt{bb}}$	Number of ballot boxes of an election event
$\mathtt{N}_{\mathtt{C}}$	Number of confirmed votes in a specific ballot box
$\hat{\mathtt{N}}_{\mathtt{C}}$	Number of mixed votes including trivial encryptions
$\mathtt{N}_{\mathtt{E}}$	Number of eligible voters of a specific verification card set
n	Number of voting options for a given verification card set
$n_{\mathtt{max}}$	Maximum number of voting options across all verification card sets
n_{sup}	Maximum supported number of voting options
$n_{\mathtt{total}}$	Number of distinct voting options across all verification card sets

ψ	Allowed number of selections for a given verification card set
$\psi_{\mathtt{max}}$	Maximum number of selections across all verification card sets
$\psi_{\mathtt{sup}}$	Maximum supported number of selections
$\mathbb{P}$	Set of prime numbers
$\mathbf{p}$	Vector of small prime group members
$\tilde{\mathbf{p}}$	Encoded voting options $(\tilde{\mathrm{p}}_1, \ldots, \tilde{\mathrm{p}}_n),\ \tilde{\mathrm{p}}_k \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$
p	Encryption group modulus
q	Encryption group cardinality $\mathsf{s.t.}\ p = 2q + 1$
$\boldsymbol{\sigma}$	Semantic information $(\sigma_0, \ldots, \sigma_{n-1}),\ \sigma_k \in \mathbb{A}_{UCS}{}^*$
$\tilde{\mathbf{v}}$	Actual voting options $(\mathrm{v}_0, \ldots, \mathrm{v}_{n-1}),\ \mathrm{v}_k \in \mathcal{T}_1^{50}$
$\mathtt{vcs}$	Verification card set ID
$\mathbf{vcs}$	Vector of verification card set IDs of an election event
$\mathbf{w}_{\mathtt{id}}$	Voter's encoded write-ins $(\mathrm{w}_{\mathtt{id},0}, \ldots, \mathrm{w}_{\mathtt{id},\delta-2})$
$\mathbb{Z}_q$	Set of integers modulo q
$\top$	Truth value true or successful termination
$\bot$	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 [3].
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 [4].
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 [8]:

- 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) [1] 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 The Role of the Verifier

A verifiable e-voting system requires a verifiable process and a verification software—the *verifier*— to verify the cryptographic evidence using data published on a private or public bulletin board [6]. The specification and development of the verifier should go hand in hand with the e-voting solution, and the verifier challenges and extensively tests a protocol run [5]. Therefore, the Ordinance on Electronic Voting (VEleS) [1] defines the role of the auditors and their technical aid.

> [VEleS art. 2(h)]: *auditor* means a person who checks on behalf of the canton that the ballot is correctly conducted.

> [VEleS art. 5 para. 3(b)]: The auditors evaluate the proof in an observable procedure; to do so, they must use *technical aids* that are independent of and isolated from the rest of the system.

For the rest of the document, we will no longer distinguish between auditors and their technical aid; we refer to *the verifier* as both the auditor and the software used by this auditor and assume that the auditor and the technical aid are trustworthy. However, the auditors must perform certain checks manually which cannot be executed by software—see section 2.5. There may be multiple auditors, and an auditor may use various technical aids. The technical annex of the Ordinance on Electronic Voting states that at least one of the auditors and one of the technical aids is trustworthy for universal verifiability.

> [VEleS Annex 2.9.2.2]: The following system participants may be considered trustworthy
> - [...]
> - one auditor in any group, leaving open which auditor it is
> - one technical aid from a trustworthy auditor, leaving open which aid it is.

In principle, everybody could become an auditor and could check an election event's proofs and data by themselves. The auditors *represent* voters in the sense of universal verifiability as elaborated in the Federal Chancellerys' explanatory report [2].

> [Explanatory Report, Sec 4.2.1]: The use of auditors promotes transparency. Voters should be able to assume that auditors will draw attention to possible irregularities.

> [Explanatory Report, Sec 4.2.1]: With universal verifiability, manipulations in the infrastructure can be detected. Unlike individual verifiability, it does not necessarily have to be offered to voters. Instead, auditors can be employed to apply universal verifiability.

> [VEleS art. 2(i)]: Infrastructure means hardware, software [...], network elements, premises, services and equipment of any nature at any operating bodies that are required for the secure operation of electronic voting.

The auditors are subject to the following requirement:

> [VEleS Annex 8.14]: The auditors should be suitably informed about and trained in the processes that determine the accuracy of the result, the preservation of voting secrecy and the avoidance of premature results (for example key generation, printing the voting papers, decryption and tallying). They must be able to understand the essential aspects of the processes and their significance.

However, the exact selection of auditors is a cantonal responsibility and out of the scope of this document:

> [VEleS art. 14]: Responsibility for running the ballot with electronic voting correctly.
> The canton shall appoint a body at cantonal level that bears overall responsibility, and for the following tasks in particular. [...]
> h. supporting and instructing the auditors.

Swiss Post releases its verifier under a permissive *open-source license*: strengthening the independence and trustworthiness of the verifiability of the system and complying with the Ordinance:

> [VEleS Annex 3.18]: The software for the auditors' technical aids must be obtained from a different system developer from the one who developed the main part of the software for the other system components. The publication of the software for the technical aid under a licence that meets the criteria for open source software may justify an exception.

1.2 Conventions

The verifier specification follows the same conventions specified in the `Conventions` section of the system specification [11].

1.3 Context and Input Variables

The verifier specification uses the same concept of context and input variables as detailed in the `Context, State, and Input Variables` section of the system specification [11]. We slightly deviate from the system specification for the consistency verifications. These verifications are very simple and as a consequence all variables are treated as input, without differentiating between context and input variables.

2 The Verifier in the Swiss Post Voting System

2.1 Structure of the Document

We distinguish two runs of the verifier according to their objectives and context. The first run happens after the configuration phase of the election event and before the voters can start submitting encrypted votes. The objective is to ensure that all parameters are valid and have been generated in a way that protects the security goals of the system.
The second run is carried out after the tally phase. At this stage, the verifications should ensure that the security properties of the system have been upheld and that the election outcome reflects the combination of the ballots submitted by the voters. This implication is further discussed in [10].
The verifications required for the first run (VerifyConfigPhase) are presented in section 3, while those needed for the final run (VerifyTally) are detailed in section 4.
Each verification run executes all verifications we document in the corresponding section and the auditors must check that every verification ran successfully. Otherwise, the auditors stop the process and a detailed analysis of the failed verification takes place. We omit a detailed pseudocode of VerifyConfigPhase and VerifyTally since their only purpose is to execute all verifications of the corresponding run.

2.2 Verification Categories

Both verifier runs contain verifications for various categories, which we will identify as proposed by Haenni et al. in [6]. Table 1 shows these categories.

Category	Description
Evidence	Are the cryptographic evidence contained in the election data all valid? Do they provide the necessary evidence to infer the correctness of corresponding protocol steps?
Authenticity	Can the data elements be linked unambiguously to the party authorized to create them?
Consistency	Are related data items consistent to each other?
Integrity	Do all data elements correspond to the specification? Are they all within the specified ranges?
Completeness	Do the data elements allow a complete verification chain?

Tab. 1: Verification categories and their description.

This document provides the pseudocode algorithms for the verification of the cryptographic *evidence*. The *authenticity* checks are based on digital signatures, with each party being identified and their signing keys known before the election starts. When necessary, the verifier specification also highlights important *consistency* checks. Since we use a mathematically precise pseudocode specification, most of the *integrity* checks consist of validating the input ranges and preconditions. Furthermore, we base our verifier specification on the computational proof of complete verifiability and privacy [10], thereby making sure that our verifications are *complete* and ensure the necessary security objectives, provided the verifier has received all elements listed in this document.

2.3 Channel Security and Control Component Authenticity

A meaningful verification of election event data must include a check that the protocol's run involved the actual honest components. Otherwise, the adversary could impersonate protocol participants — undermining verifiability and vote secrecy.
Recall that our trust model considers the setup component and one out of four control components trustworthy. The Ordinance's explanatory report elaborates on the term *trustworthy*.

> [Explanatory Report, Sec 5.2.2]: Cryptographic protocols make it possible to reduce to a minimum the number of elements that an attacker would have to control in order to manipulate votes without being detected or violate voter secrecy. Measures to prevent an attacker from taking control of an element can therefore focus on a limited number of elements. These elements are particularly worthy of protection and, ideally, can also be protected particularly effectively. Such elements found under Numbers 2.1 and 2.2 'System participants' and 'Communication channels' are referred to as 'trustworthy'. This may seem surprising at first glance: why is an element that is particularly worthy of protection called 'trustworthy'? The reason lies in the fact that cryptographic protocols are not aimed at protecting those elements. The designation 'trustworthy' signals to authors and readers of the document in which the cryptographic protocol is specified that they do not need to worry about possible attacks in which an attacker takes control of these elements. By being trustworthy, system participants *refuse* to cooperate with an attacker. The protocol must be defined in such a way that, as long as the trustworthy system participants adhere to the protocol, the attacker will not succeed even if they bring the remaining non-trustworthy system participants under control. The use of the term is based on the literature.

All elements received by the verifier must be signed by the expected party ~~, using the GenSignature~~ with the GenSignature algorithm from the crypto primitives specification~~ [9]~~ [9] or the GenXMLSignature algorithm from the system specification[11]. The exact data being signed as well as the additional context data used for the signature are specified in the `Channel Security` section of the system specification [11], and are reprised in this document for each of the corresponding *authenticity* checks.
In order to be able to verify the signatures (using algorithm ~~VerifySignature from the crypto primitives~~ VerifySignature from the crypto primitives specification or the VerifyXMLSignature algorithm from the system specification), the verifier must first be made aware of signature keys of each of the parties. As such, each party (*i.e.* control components, tally component and setup component) designates a responsible person that transmits their certificate out-of-band to the auditor during a certificate ceremony. The certificates are then imported into the verifier's keystore after having been verified as per section `Importing a Trusted Certificate` from the crypto primitives specification. Figure 1 highlights the verification procedure.

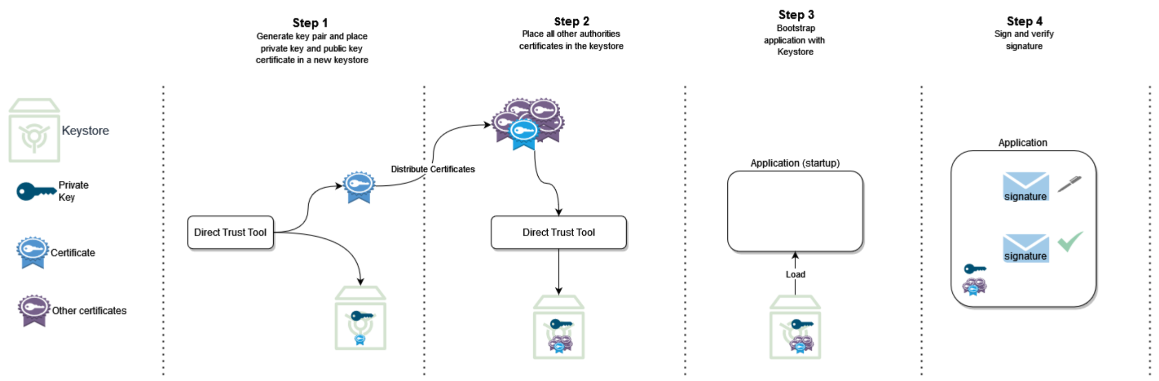
Step 1
Generate key pair and place private key and public key certificate in a new keystore
Step 2
Place all other authorities certificates in the keystore
Step 3
Bootstrap application with Keystore
Step 4
Sign and verify signature
Keystore
Private Key
Certificate
Other certificates
Direct Trust Tool
Distribute Certificates
Direct Trust Tool
Application (startup)
Load
Application
signature
signature

Fig. 1: The control components generate a certificate, share it out of band with the auditors, who import them in their keystore, thus enabling them to directly verify the authenticity of signed messages.

The certificate ceremony allows the auditors to verify that the data it received originated from the expected party.

2.4 Streamable Symmetric Encryption and Decryption for Dataset Security

The verifier's datasets are symmetrically encrypted as an additional security mechanism. Given the potentially huge size of the verifier's dataset, the verifier requires a decryption mechanism that can handle datasets too large to fit entirely in memory. This is an additional security layer since the data for the verifier (for instance the votes) are already encrypted. We refer to the system specification section `Streamable Symmetric Encryption and Decryption for Dataset Security` for the details on how to decrypt the verifier datasets.

2.5 Manual Checks by the Auditors

Certain domain-specific verifications cannot be run by software; they must be checked manually by the auditors operating the verifier. The explanatory report highlights that certain manual checks are necessary [2]:

> [Explanatory Report, Sec 4.2.1]: The auditors must ascertain that the number of authentication credentials corresponds to the (official) number of authorised voters.

The verifier must create a document that can be easily understood and signed by auditors. This document should clearly indicate the datasets being used for the audit by printing the hash value of the datasets.
We assume that the auditors have access to the necessary information regarding the expected, correct configuration of the election event. This includes knowledge of the following parameters:

Parameter	**Comment**
Name and date of the election event	Public information
Number of elections and votes	Public information. In this context, the term 'vote' refers specifically to a group of referendum-style questions.
Number of productive and test ballot boxes	The number of ballot boxes relates to the number of counting circles in a canton and is known to the auditors
Number of eligible real and test voters	Can be learned from the electoral roll

Tab. 2: Overview of the auditor's knowledge of the election event parameters

The auditors can learn the information in table 2 from publicly available sources, compare it against data from previous election events, or confirm it against the actual electoral roll.
The verifier will present a document to the auditors, displaying the values discussed above, which are sourced from the canton's Election Event Configuration, as described in section 3.1. Before presenting the document, the verifier performs authenticity and consistency checks on the canton's Election Event Configuration in sections 3.2 and 3.3 to ensure that the information is authentic and consistent. Only if all relevant verifications are successful, and the auditors have manually verified the information, will the auditors sign the document to confirm its accuracy.
The Election Event Configuration—signed by the canton (see section 3.2)—contains additional information about the election event, such as the precise wording of the questions, the names of the candidates and lists, and an identifier for each voting option. While the auditors can assume that this additional information is correct, it is still recommended that they manually verify the accuracy of this information by cross-checking it against other available sources.

The below pseudocode indicates the checks that the auditors must perform manually.

Verification 0.01 ManualChecksByAuditors

```
Context:
    The protocol participants' certificates—received via an out-of-band channel (section 2.3).
    When executing VerifyTally: the context dataset's hash value from the VerifyConfigPhase.
    The auditors' knowledge of the election event parameters indicated in table 2.
Input:
    A verifier's execution in the VerifyConfigPhase or VerifyTally.
    The verifier's report(s) indicating:
        - the certificate fingerprints
        - the name and date of the election event
        - the number of elections and votes
        - the number of productive and test ballot boxes
        - the number of eligible real and test voters
        - the execution status of the VerifyConfigPhase or VerifyTally verifications
Operation:
  1: When executing VerifyTally, check that the ~~context dataset's hash equals the one from the~~ hash of the context dataset equals the value recorded during VerifyConfigPhase, and that the hash of the verifier's eCH-0222 XML equals the hash of the eCH-0222 XML imported into the system for consolidation and publication.
  2: Check that the certificates' fingerprints match the expected ones.
  3: Check that the name and date of the election event correspond to the expected ones.
  4: Check that the number of elections and votes corresponds to the expected one.
  5: Check that the number of productive and test ballot boxes corresponds to the expected one.
  6: Check that the number of real and test voters corresponds to the expected one.
  7: Check that all expected verifications executed successfully.
                                          ▷ See crypto primitives specification
Output:
    ⊤ if the verification succeeds, ⊥ otherwise.
```

In the VerifyTally phase, the auditors must have access to the signed audit document from the VerifyConfigPhase.
Once VerifyTally has been successfully executed, the auditors must review and acknowledge the election results. This process involves spot checking the results to ensure their plausibility and accuracy. ~~To further increase transparency and accountability~~
Besides the check in ManualChecksByAuditors that the correct version of eCH-0222 XML file is used, the auditors should record ~~some of the election results, which can~~ selected election results. These records can later be compared against the ~~official published results at a later time. This record-keeping can help to identify any potential discrepanciesand ensure that the election results are reliable and trustworthy.~~

~~Finally, the verified election result is provided to the system responsible for consolidating and publishing the election results. It is critical that only results verified by the auditors are forwarded for consolidation and publication~~officially published results to detect any discrepancies.

2.6 Election Event Context

We refer the reader to the system specification, which explains the election event context and that some algorithms are executed per election event, per verification card set, per ballot box, or voting card.

2.7 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 Setup Verification - VerifyConfigPhase

This section defines the verifications that need to be performed before the voters may be allowed to start voting. This is meant to ensure that the election event has been configured correctly and according to the specification, with all parties having provided the required proof for the data they generated. In case an error appears during verification of the setup phase, it must be determined if the cause is a misconfiguration of the verifier or if further investigation is needed. In the first case, the verifier may simply be configured and run anew. In the second case, once the root cause has been identified and fixed, the setup phase must be restarted from scratch.

3.1 Setup - Completeness

The VerifyConfigPhase requires a complete context (table 3)~~and setup dataset (??)~~. The implementation must ensure that the ~~datasets~~ dataset contains the expected files in the correct file structure.
The placeholders represent the different identifiers present in the paths. To be more specific, the placeholder `${j}` represents the control component index, the placeholder `${vcs}` represents the verification card set identifier~~, and the placeholder `${chunkId}` represents the chunk identifier~~. The different consistency verifications ensure that these indices and identifiers are consistent to the information in the ~~election event context~~Election Event Context.

Description	Path
Election Event Context	`context/electionEventContextPayload.json`
Election Event Configuration	`context/configuration-anonymized.xml`
Online Control Component Public Keys	`context/controlComponentPublicKeysPayload.${j}.json`
Setup Component Public Keys	`context/setupComponentPublicKeysPayload.json`
Setup Component Tally Data	`context/verificationCardSets/${vcs}/setupComponentTallyDataPayload.json`

Tab. 3: The contents of the context dataset. The context dataset is used both in VerifyConfigPhase and VerifyTally

~~Setup Component Verification Data `setup/verificationCardSets/${vcs}/setupComponentVerificat`~~
~~Control Component Code Shares `setup/verificationCardSets/${vcs}/controlComponentCodeShare`~~
~~The contents of the setup dataset. The setup dataset is used only in VerifyConfigPhase~~

Verification 1.01 VerifySetupCompleteness

Input:
The context ~~and setup datasets~~dataset.

Operation:
1: ~~Check that the datasets contain~~ Verify that the dataset contains all required elements from table 3~~and ??~~.

Output:
$\top$ if the verification succeeds, $\perp$ otherwise.

3.2 Setup - Authenticity

Table 4 provides an overview of the authenticity checks for the setup verification. Each element corresponds to an entry in table 3~~or ??~~, and provides the details of what should be given as input to the `VerifySignature` algorithm.

Message Name	Signer	Message Content	Context Data
ElectionEventContext	Setup Comp.	~~$(p, q, g, \mathtt{seed}, \mathbf{p}, n_{\mathtt{max}}, \psi_{\mathtt{max}}$~~ ~~$\delta_{\mathtt{max}}, \{\mathtt{vcs}, \mathsf{pTable}, \mathtt{N_E}\}^{\forall \mathtt{vcs}})$~~ ElectionEventContextPayload, see system specification table 18	(“`election event context`”, `ee`)
CantonConfig	Canton	configuration XML, see system specification table 15	~~(“`configuration`”)~~ -
ControlComponentPublicKeys	Online CC$_j$	~~$(\mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathsf{pkCCR},j}, \mathsf{EL}_{\mathsf{pk},j}, \pi_{\mathsf{ELpk},j})$~~ ControlComponentPublicKeysPayload, see system specification table 18	(“`OnlineCC keys`”, j, `ee`)
	Setup Comp.	~~$(\{j, \mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathsf{pkCCR},j}, \mathsf{EL}_{\mathsf{pk},j}, \pi_{\mathsf{ELpk},j}\}_{j=1}^{4}, \mathsf{EB}_{\mathsf{pk}}, \pi_{\mathsf{EB}}, \mathsf{EL}_{\mathsf{pk}}, \mathbf{pk}_{\mathrm{CCR}})$~~ SetupComponentPublicKeysPayload, see system specification table 18	(“`public keys`”, “`setup`”, `ee`)
~~SetupComponentVerificationData~~	~~Setup Comp.~~	~~$(\{\mathtt{vc_{id}}, \mathtt{K_{id}}, \mathtt{c_{pCC,id}}, \mathtt{c_{ck,id}}\}_{id=0}^{\mathtt{N_E}-1}, \mathtt{L_{pCC}})$~~	~~(“`verification data`”, `ee`, `vcs`)~~
~~ControlComponentCodeShares~~	~~Online CC$_j$~~	~~$(\{\mathtt{vc_{id}}, \mathtt{K}_{j,id}, \mathtt{Kc}_{j,id}, \mathtt{c}_{\mathtt{expPCC},j,id}, \mathtt{c}_{\mathtt{expCK},j,id}, \pi_{\mathsf{expPCC},j,id}, \pi_{\mathsf{expCK},j,id}\}_{id=0}^{\mathtt{N_E}-1})$~~	~~(“`encrypted code shares`”, j, `ee`, `vcs`)~~
SetupComponentPublicKeys			
SetupComponentTallyData	Setup Comp.	~~$(\mathbf{vc}, \mathbf{K})$~~ SetupComponentTallyDataPayload, see system specification table 18	(“`tally data`”, `ee`, `vcs`)

Tab. 4: Overview of the authenticity checks for the setup verification

The ~~configuration XML above, as well as the two other XML files mentioned in table 7, are signed according to the following high-level process: starting from the root element of the XML file, each complexType element is represented as a nested vector of values within the domain accepted by RecursiveHash, within such complexType, elements are taken in the order in which they are defined in the XSD, if an element is optional and absent, the string~~ configuration XML is signed with the algorithm GenXMLSignature as described in the section ~~"no <tokenname> value"is hashed with the value of tokenname being replaced with the token name, to prevent trivial collisions in case of several optional elements following each other, 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, ...) and also date-like types are represented as strings, the signature field itself must be ignored.~~ Channel Security of the system specification [11].

Verification 2.01 VerifySignatureCantonConfig

Context:

The trust store containing the system's certificates.

Input:

~~The message CantonConfig from table 4The signature $s \in \mathcal{B}^*$~~ Signed configuration XML from table 4.

Operation:

```
1: ~~VerifySignature(“canton”, configuration XML, (“configuration”), $s$)~~
       VerifyXMLSignature(configuration XML, $\mathsf{pk}_{\mathtt{canton}}$)      ▷ see system specification. The
   canton's public key $\mathsf{pk}_{\mathtt{canton}}$ is extracted from the trust store.
```

Output:

~~$\top$ if the~~ $\top$ if verification succeeds, ~~$\bot$~~ $\bot$ otherwise.

Verification 2.02 VerifySignatureSetupComponentPublicKeys

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentPublicKeys from table 4

The signature $s \in \mathcal{B}^*$

Operation: ~~VerifySignature(“sdm_config”,~~

```
~~$(\{j, \mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathsf{pkCCR},j}, \mathsf{EL}_{\mathsf{pk},j}, \pi_{\mathsf{ELpk},j}\}_{j=1}^{4}, \mathsf{EB}_{\mathsf{pk}}, \pi_{\mathsf{EB}}, \mathsf{EL}_{\mathsf{pk}}, \mathbf{pk}_{\mathrm{CCR}})$,~~
~~(“public keys”, “setup”, ee), $s$)~~            ▷ For all algorithms see the crypto primitives
specification                                    ▷ We use nested structures in this algorithm
1: for $j \in [1, 4]$ do
2:     $h_{\mathsf{pk},j} \leftarrow (j, \mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathsf{pkCCR},j}, \mathsf{EL}_{\mathsf{pk},j}, \pi_{\mathsf{ELpk},j})$
3: end for
4: $h_{\mathsf{pk}} \leftarrow (h_{\mathsf{pk},1}, h_{\mathsf{pk},2}, h_{\mathsf{pk},3}, h_{\mathsf{pk},4})$
5: $\mathtt{hPublicKeys} \leftarrow (h_{\mathsf{pk}}, \mathsf{EB}_{\mathsf{pk}}, \pi_{\mathsf{EB}}, \mathsf{EL}_{\mathsf{pk}}, \mathbf{pk}_{\mathrm{CCR}})$
6: $h \leftarrow \big((p, q, g), \mathtt{ee}, \mathtt{hPublicKeys}\big)$
7: $d \leftarrow$ Base64Encode(RecursiveHash($h$))
8: VerifySignature(“sdm_config”, $d$, (“public keys”, “setup”, ee), $s$)
```

Output:

$\top$ if the verification succeeds, $\bot$ otherwise.

Test values for the verification 2.02 are provided in verify-signature-setup-component-public-keys.json.

Verification 2.03 VerifySignatureControlComponentPublicKeys

Context:

The trust store containing the system's certificates

Input:

The message ControlComponentPublicKeys from table 4
The signature $s \in \mathcal{B}^*$

Operation: ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm

```
1: ~~$\mathsf{VerifySignature}(\text{“}\texttt{control\_component\_j}\text{”}, (\mathbf{pk}_{\mathrm{CCR}_j}, \mathsf{EL}_{\mathsf{pk},j}, \boldsymbol{\pi}_{\mathsf{ELpk},j}), (\text{“}\texttt{OnlineCC keys}\text{”}, j, \texttt{ee}), s)$~~ $\texttt{hPublicKeys} \leftarrow (j, \mathbf{pk}_{\mathrm{CCR}_j}, \boldsymbol{\pi}_{\mathsf{pkCCR},j}, \mathsf{EL}_{\mathsf{pk},j}, \boldsymbol{\pi}_{\mathsf{ELpk},j})$
2: $h \leftarrow \big((p, q, g), \texttt{ee}, \texttt{hPublicKeys}\big)$
3: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
4: $\mathsf{VerifySignature}(\text{“}\texttt{control\_component\_j}\text{”}, d, (\text{“}\texttt{OnlineCC keys}\text{”}, j, \texttt{ee}), s)$
```

Output:

$\top$ if the verification succeeds, $\bot$ otherwise.

Test values for the verification 2.03 are provided in
verify-signature-control-component-public-keys.json.

Verification 2.04 VerifySignatureSetupComponentTallyData

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentTallyData from table 4
The signature $s \in \mathcal{B}^*$

Operation: ~~$\mathsf{VerifySignature}(\text{“}\texttt{sdm\_config}\text{”}, (\texttt{vc}, \mathbf{K}), (\text{“}\texttt{tally data}\text{”}, \texttt{ee}, \texttt{vcs}), s)$~~ ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm

```
1: $h \leftarrow \big((p, q, g), \texttt{ee}, \texttt{vcs}, \mathbf{vc}, \texttt{ballotBoxDefaultTitle}, \mathbf{K}\big)$
2: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
3: $\mathsf{VerifySignature}(\text{“}\texttt{sdm\_config}\text{”}, d, (\text{“}\texttt{tally data}\text{”}, \texttt{ee}, \texttt{vcs}), s)$
```

Output:

$\top$ if the verification succeeds, $\bot$ otherwise.

Test values for the verification 2.04 are provided in
verify-signature-setup-component-tally-data.json.

Verification 2.05 VerifySignatureElectionEventContext

Context:

The trust store containing the system's certificates

Input:

The message ElectionEventContext from table 4

The signature $s \in \mathcal{B}^*$

Operation: ~~$\mathsf{VerifySignature}(\text{“}\texttt{sdm\_config}\text{”}, \mathsf{ElectionEventContext}, (\text{“}\texttt{election event context}\text{”}, \texttt{ee}), s)$~~

```
        ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm
1: for vcs ∈ vcs do
2:     $h_{\texttt{pTable,j}} \leftarrow \left(\left((\mathrm{v}_0, \tilde{\mathrm{p}}_0, \sigma_0, \tau_0), \ldots, (\mathrm{v}_{n-1}, \tilde{\mathrm{p}}_{n-1}, \sigma_{n-1}, \tau_{n-1})\right)\right)$
3:     $h_{\texttt{vcs,j}} \leftarrow (\texttt{vcs}, \texttt{vcsAlias}, \texttt{vcsDesc}, \texttt{bb}, t_{\mathrm{s,bb}}, t_{\mathrm{f,bb}}, \texttt{testBallotBox}, \mathrm{N_E}, \texttt{gracePeriod}, h_{\texttt{pTable,j}}, \mathbf{DoI})$
4: end for
5: $h_{\texttt{vcs}} \leftarrow (h_{\texttt{vcs},0}, h_{\texttt{vcs},1}, \ldots, h_{\texttt{vcs},\mathrm{N_{bb}}-1})$
6: $\texttt{hContext} \leftarrow \left((p, q, g), \texttt{ee}, \texttt{eeAlias}, \texttt{eeDesc}, h_{\texttt{vcs}}, t_{\mathrm{s,ee}}, t_{\mathrm{f,ee}}, n_{\texttt{max}}, \psi_{\texttt{max}}, \delta_{\texttt{max}}\right)$
7: $h \leftarrow \left((p, q, g), \texttt{seed}, \mathbf{p}, \texttt{hContext}, \texttt{tenantId}\right)$
8: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
9: $\mathsf{VerifySignature}(\text{“}\texttt{sdm\_config}\text{”}, d, (\text{“}\texttt{election event context}\text{”}, \texttt{ee}), s)$
```

Output:

$\top$ if the verification succeeds, $\perp$ otherwise.

~~We verify the signature of the setup component verification data and the control component code shares as algorithms within the verification ??. The verification of this signature requires the deserialization of huge messages, which can be extremely time-consuming for large election events. By including this authenticity check within theverification ??, we ensure that the verifier deserializes each data item only once.~~

~~The trust store containing the system's certificates~~

~~The message SetupComponentVerificationData from table 4 The signature $\mathsf{s_{SCVD}} \in \mathcal{B}^*$~~

~~$\mathsf{VerifySignature}(\text{“}\texttt{sdm\_config}\text{”}, (\{\texttt{vc}_{\texttt{id}}, \mathsf{K}_{\texttt{id}}, \mathsf{c}_{\mathsf{pCC},\texttt{id}}, \mathsf{c}_{\mathsf{ck},\texttt{id}}\}_{id=0}^{\mathrm{N_E}-1}, \mathsf{L}_{\mathsf{pCC}}),$~~
~~$(\text{“}\texttt{verification data}\text{”}, \texttt{ee}, \texttt{vcs}), \mathsf{s_{SCVD}})$~~

~~$\top$ if the verification succeeds, $\perp$ otherwise.~~

~~The trust store containing the system's certificates~~

~~The message ControlComponentCodeShares from table 4 The signature $\mathsf{s_{CCCS}} \in \mathcal{B}^*$~~

~~$\mathsf{VerifySignature}(\text{“}\texttt{control\_component\_j}\text{”},$~~
~~$(\{\texttt{vc}_{\texttt{id}}, \mathsf{K}_{j,id}, \mathsf{Kc}_{j,id}, \mathsf{c}_{\mathsf{expPCC},j,id}, \mathsf{c}_{\mathsf{expCK},j,id}, \pi_{\mathsf{expPCC},j,id}, \pi_{\mathsf{expCK},j,id}\}_{id=0}^{\mathrm{N_E}-1}),$~~
~~$(\text{“}\texttt{encrypted code shares}\text{”}, j, \texttt{ee}, \texttt{vcs}), \mathsf{s_{CCCS}})$~~ Test values for the verification 2.05 are provided in verify-signature-election-event-context.json.

~~$\top$ if the verification succeeds, $\perp$ otherwise.~~

3.3 Setup - Consistency

For setup - consistency we have the following verifications~~3.01-??~~.

Verification 3.01 VerifyEncryptionGroupConsistency

Input:

The encryption group parameters included in the following files from table 3~~and ??~~:
- Election Event Context
- Online Control Components Public Keys ▷ 1 per component
- Setup Component ~~Verification Data - Control Component Code Shares - Setup Component~~ Tally Data ▷ 1 per verification card set

Operation:

Verify that the encryption group parameters are consistent across all files.

Output:

⊤ if all encryption group parameters are identical, ⊥ otherwise.

Verification 3.02 VerifyNodeIdsConsistency

Input:

The ~~files~~ node IDs included in the following file from table 3:
- Online Control Component Public Keys ▷ 1 per component

Operation:

1: Verify that the node IDs are consistent across all files, i.e. that all files have exactly one contribution from each node.

Output:

⊤ if the node IDs are consistent, ⊥ otherwise.

Verification 3.03 VerifyFileNameNodeIdsConsistency

Input:

The following file from the context ~~and setup datasets in table 3and ??~~ dataset in table 3:
- Online Control Component Public Keys ▷ 1 per component

Operation:

~~Check~~ Verify that the file names match the paths in the directory of the dataset.

Output:

⊤ if the file names are consistent, ⊥ otherwise.

Verification 3.04 VerifyElectionEventIdConsistency

Input:

The election event ID included in the files from table 3.

Operation:

```
1: Verify that the election event ID ee is consistent across all files.
```

Output:

$\top$ if the election event ID is consistent, $\bot$ otherwise.

Verification 3.05 VerifyVerificationCardSetIdsConsistency

Input:

The verification card set IDs included in the files from table 3.

Operation:

```
1: Verify that the verification card set IDs vcs are consistent across all files.
2: Verify that the path names containing the verification card set ID in the context dataset
   match the verification card set ID within the files.
```

Output:

$\top$ if the verification card set IDs are consistent, $\bot$ otherwise.

Verification 3.06 VerifyFileNameVerificationCardSetIdsConsistency

Input:

The Election Event Context from table 3.
The paths of the context dataset (table 3).

Operation:

```
1: Verify that path names of the context dataset match the list of verification card set IDs
   in the Election Event Context.
```

Output:

$\top$ if the verification card set IDs between the dataset and the Election Event Context are consistent, $\bot$ otherwise.

Verification 3.07 VerifyVerificationCardIdsConsistency

Input:

The following files from table 3:
- Election Event Context
- Setup Component Tally Data

Operation:

```
1: Verify that there are no duplicate verification card IDs $\mathtt{vc_{id}}$ across all files.
2: Verify that the number of verification card IDs matches the number of eligible voters for all verification card sets.
```

Output:

$\top$ if all verification card IDs are consistent, $\perp$ otherwise.

Verification 3.08 VerifyCCRChoiceReturnCodesPublicKeyConsistency

Input:

The CCR Choice Return Codes encryption public keys ($\mathbf{pk}_{\mathrm{CCR}_j}$) included in the following files from table 3:
- Online Control Component Public Keys ▷ 1 per component
- Setup Component Public Keys

Operation:

```
1: for $j \in [1, 4]$ do
2:     $\mathrm{ok}_j \leftarrow$ the CCR Choice Return Codes encryption public keys for control component $j$ are identical from both sources
3: end for
```

Output:

$\top$ if all keys are identical, $\perp$ otherwise.

Verification 3.09 VerifyCCMElectionPublicKeyConsistency

Input:

The CCM election public keys ($\mathbf{pk}_{\mathrm{CCR}_j}$) included in the following files from table 3:
- Online Control Component Public Keys ▷ 1 per component
- Setup Component Public Keys

Operation:

```
1: for $j \in [1, 4]$ do
2:     $\mathrm{ok}_j \leftarrow$ the CCM election public key for control component $j$ is identical from both sources
3: end for
```

Output:

$\top$ if all keys are identical, $\perp$ otherwise.

Verification 3.10 VerifyCCMAndCCRSchnorrProofsConsistency

Input:

```
- Online Control Component Public Keys          ▷ 1 per component
- Setup Component Public Keys
```

Operation:

```
1: Verify that the CCM and CCR Schnorr proofs are identical from both sources.
```

Output:

$\top$ if all keys are identical, $\bot$ otherwise.

Verification 3.11 VerifyChoiceReturnCodesPublicKeyConsistency

Input:

```
- Online Control Component Public Keys          ▷ 1 per component
- Setup Component Public Keys
```

Operation:

```
1: Verify that $\mathbf{pk}_{\mathrm{CCR}} = \prod_{j=1}^{4} \mathbf{pk}_{\mathrm{CCR}_j} \pmod p$    ▷ $\mathbf{pk}_{\mathrm{CCR}_j}$ taken from the Online Control Component Public Keys
```

Output:

$\top$ if the Choice Return Codes encryption public key $\mathbf{pk}_{\mathrm{CCR}}$ was correctly combined, $\bot$ otherwise.

Verification 3.12 VerifyElectionPublicKeyConsistency

Input:

```
- Online Control Component Public Keys          ▷ 1 per component
- Setup Component Public Keys
```

Operation:

```
1: Verify that $\mathtt{EL}_{\mathtt{pk}} = \mathtt{EB}_{\mathtt{pk}} \cdot \prod_{j=1}^{4} \mathtt{EL}_{\mathtt{pk},j} \pmod p$    ▷ $\mathtt{EL}_{\mathtt{pk},j}$ taken from the Online Control Component Public Keys
```

Output:

$\top$ if the election public key $\mathtt{EL}_{\mathtt{pk}}$ was correctly combined, $\bot$ otherwise.

Verification 3.13 must derive the expected voting options from the ~~configuration XML~~configuration XML. See table 3 (Election Event Configuration). The ~~configuration XML~~ configuration XML describes either elections or referendum-style votes. Table 5 describes how to derive the actual voting options and semantic information, depending on which kind of election or referendum-style vote is described in the ~~configuration XML~~configuration XML. Also the correctness information can be constructed. See section `Encoding of Voting Options` of the system specification for how to construct the correctness information.

Type	Actual Voting Option	Semantic Information
election/ Lists	election identifier | list identifier	"NON_BLANK" or "BLANK" | list description
election/ Candidates	election identifier | candidate identifier | candidate accumulation	"NON_BLANK" | family name | ~~first name |~~ call name | date of birth
election / Blank candidate position	election identifier | blank candidate position identifier	"BLANK" | EMPTY_CANDIDATE_POSITION-[position number]
election / Write-in positions	election identifier | write-in position identifier	"WRITE-IN" | WRITE_IN_POSITION-[position number]
vote / Standard questions	question identifier | answer identifier	"NON_BLANK" or "BLANK" | question | answer
vote / Tie-break questions	question identifier | answer identifier	"NON_BLANK" or "BLANK" | question | answer

Tab. 5: Overview of the different voting option types necessary to validate the pTable.

Verification 3.13 VerifyPrimesMappingTableConsistency

Input:

~~-~~Election Event Context ▷ See table 3
~~-~~Election Event Configuration configuration XML ▷ See table 3

Operation:

1: Verify that the same actual voting option v_i maps to the same encoded voting option $\tilde{p}_i$, that the same actual voting option v_i maps to the same semantic information σ_i and that the same actual voting option v_i maps to the same correctness information τ_i in all verification card sets.
2: Verify that the actual voting options, semantic information and correctness information in the pTable correspond to the ~~configuration XML~~configuration XML, see table 5.
3: Verify that the number of entries in the pTable correspond to the ~~configuration XML~~ configuration XML taking into account possible accumulation of candidates.

Output:

⊤ if the primes mapping tables pTable in all verification card sets are consistent and correspond to the ~~configuration XML, ⊥ otherwise.~~

~~The election event ID included in the files from table 3 and ??.~~

~~Verify that the election event ID ee is consistent across all files.~~

~~⊤ if the election event ID is consistent, ⊥ otherwise.~~

~~The verification card set IDs included in the files from table 3 and ??.~~

~~Verify that the verification card set IDs vcs are consistent across all files. Verify that the path names containing the verification card set ID in the context and setup dataset match the verification card set ID within the files.~~

~~⊤ if the verification card set IDs are consistent, ⊥ otherwise.~~

~~The Election Event Context from table 3. The paths of the context (table 3) and setup dataset (??).~~

~~Verify that path names of the context and setup dataset match the list of verification card IDs in the Election Event Context.~~

~~⊤ if the verification card set IDs between the datasets and the Election Event Context are consistent, ⊥ otherwise.~~

~~The verification card IDs included in the files from ??.~~

~~Verify that the verification card IDs vc_{id} match in content and order across all files and that there are no duplicate verification card IDs.~~

~~⊤ if all verification Card IDs are consistent and in the right order~~configuration XML, ⊥ otherwise.

Verification 3.14 VerifyTotalVotersConsistency

Input:

The following files from table 3:
- Election Event Configuration configuration XML
- Election Event Context

Operation:

1: Verify that the number of voters in the Election Event Configuration matches the sum of the number of eligible voters in all verification card sets in the Election Event Context.

Output:

$\top$ if the number of voters is consistent, $\perp$ otherwise.

~~The node IDs included in the following files from table 3 and ??: - Online Control Component Public Keys - Control Component Code Shares~~

~~Verify that the node IDs are consistent across all files, i.e. that all files have exactly one contribution from each node.~~

~~$\top$ if the node IDs are consistent, $\perp$ otherwise.~~

~~The chunk IDs included in the following files from ??: - Setup Component Verification Data - Control Component Code Shares~~

~~Verify that the chunkIDs form an uninterrupted monotonic sequence. Verify that the chunkID within the files matches the chunkID in the file name.~~

~~$\top$ if the chunkIDs are consistent, $\perp$ otherwise.~~

3.4 Setup - Integrity

All pseudocode algorithms define the domain for each input. All inputs must be verified to be in the expected domains.

3.5 Setup - Evidence

The pseudocode algorithms in this section verify the evidence generated during the setup phase. Namely, these elements demonstrate that the Swiss Post Voting System configured cryptographically sound parameters, associated each voting option to a prime number, generated the correct number of voting cards, and signed the relevant configuration information. A positive verification result of these elements indicates to the auditor that the configuration allows the system to perform reasonably secure operations and to provide meaningful cryptographic evidence. In contrast, failed verifications in VerifyConfigPhase cast doubts about the system's security properties: secure encryption of votes, unforgeability of digital signatures, and the soundness of zero-knowledge proofs rely on the proper configuration of the cryptographic parameters.

The `seed` value which is an input to Verification 5.01 must be of the correct format as defined in the system specification [11], in the section `Cryptographic Parameters and System Security Level`.

Verification 5.01 VerifyEncryptionParameters

```
Input:                         ▷ All inputs are taken from table 3 – Election Event Context
    Provided group modulus $\hat{p} \in \mathbb{P}$
    Provided group cardinality $\hat{q} \in \mathbb{P}$ s.t. $\hat{p} = 2\hat{q} + 1$
    Provided group generator $\hat{g} \in \mathbb{G}_q$
    seed $\in \mathbb{A}_{UCS}^{16}$
        ▷ The name of the election event in the format specified in the system specification
Require:
    Verify that $|\hat{p}|$ corresponds to the *standard* security level ▷ See the section *Security Level*
    in the crypto primitives specification
Operation:
1: $(p, q, g) \leftarrow$ GetEncryptionParameters(seed)        ▷ See crypto primitives specification
2: if $(p = \hat{p}) \wedge (q = \hat{q}) \wedge (g = \hat{g})$ then
3:     return $\top$
4: else
5:     return $\bot$
6: end if
Output:
    The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.
```

The verifier checks that the small primes— excluding the generator g—of a mathematical group $\mathbb{G}_q$ encode the voting options. Since the voting client multiplies all selected voting options prior to encryption, the verifier ensures that the maximum product of ψ_{sup} voting options does not exceed p to prevent modulo overflow.

Verification 5.02 VerifySmallPrimeGroupMembers

```
Context:                    ▷ All contexts are taken from table 3 – Election Event Context
    Group modulus $p \in \mathbb{P}$
    Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
    Group generator $g \in \mathbb{G}_q$
    Maximum supported number of voting options $n_{\mathsf{sup}} \in \mathbb{N}^+$
Input:
    The small prime group members in ascending order $\mathbf{p} = (\mathrm{p}_0, \ldots, \mathrm{p}_{n_{\mathsf{sup}}-1}) \in (\mathbb{G}_q \cap \mathbb{P}) \setminus \{2, 3\}, (\mathrm{p}_i < \mathrm{p}_{i+1}) \; \forall \; i \in [0, n_{\mathsf{sup}} - 1)$
Operation:
1: $\mathbf{p}' \leftarrow \mathsf{GetSmallPrimeGroupMembers}(p, q, g, n_{\mathsf{sup}})$    ▷ See crypto primitives specification
2: if $\mathbf{p}' = \mathbf{p}$ then
3:     return $\top$
4: else
5:     return $\bot$
6: end if
Output:
    The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.
```

The VerifyVotingOptions algorithm follows VerifySmallPrimeGroupMembers and assumes that the latter algorithm verified the list of small prime group members. Hence, we label this input argument as a *trusted* input.
Moreover, the system specification [11] elaborates in the section `Election Event Context` that different voters might have different voting options.
Hence, the algorithms requires as input a sorted, consolidated list of encoded voting options across all voting card sets of size $n_{\mathtt{total}}$ ($n_{\mathtt{total}} \geq n_{\mathtt{max}}$).

Verification 5.03 VerifyVotingOptions

Context: ▷ All contexts are taken from table 3 – Election Event Context

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Maximum supported number of voting options $n_{\mathtt{sup}} \in \mathbb{N}^+$
Maximum supported number of selections $\psi_{\mathtt{sup}} \in \mathbb{N}^+$

Input:

Small prime group members in ascending order $\mathbf{p} = (\mathrm{p}_0, \ldots, \mathrm{p}_{n_{\mathtt{sup}}-1}),\ \mathrm{p}_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus \{2, 3\}, (\mathrm{p}_i < \mathrm{p}_{i+1})\ \forall\, i \in [0, n_{\mathtt{sup}} - 1)$ ▷ *Trusted* input that was verified in verification 5.02
Encoded voting options in ascending order $\tilde{\mathbf{p}} = (\tilde{\mathrm{p}}_0, \ldots, \tilde{\mathrm{p}}_{n_{\mathtt{total}}-1}),\ \tilde{\mathrm{p}}_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus \{2, 3\}, (\tilde{\mathrm{p}}_i < \tilde{\mathrm{p}}_{i+1})\ \forall\, i \in [0, n_{\mathtt{total}} - 1)$ ▷ The keys of the prime mapping tables in table 3 – Tally Data

Require:

$\psi_{\mathtt{sup}} \leq n_{\mathtt{sup}}$
$0 < n_{\mathtt{total}} \leq n_{\mathtt{sup}}$

Operation:

```
1: $\mathbf{p}' \leftarrow (\mathrm{p}_0, \ldots, \mathrm{p}_{n_{\mathtt{total}}-1})$
2: if $\mathbf{p}' = \tilde{\mathbf{p}}$ then
3:     verifA $\leftarrow \top$
4: else
5:     verifA $\leftarrow \perp$
6: end if
7: verifB $\leftarrow \prod_{i=(n_{\mathtt{sup}}-\psi_{\mathtt{sup}})}^{n_{\mathtt{sup}}-1} \mathrm{p}_i < p$    ▷ The product of the $\psi_{\mathtt{sup}}$ last primes (the largest possible
   encoded vote) must be smaller than $p$
8: return verifA $\wedge$ verifB
```

Output:

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

The verifier checks the Schnorr proofs of knowledge that were generated during the configuration phase. This prevents a malicious party from providing a public key without knowing the corresponding secret key. This verification takes the Schnorr proofs from the setup component's public keys and assumes that the consistency verifications check that they correspond to the control component's data. The Schnorr proofs include the entire election event context as explained in the system specification. Thereby, it is ensured that the control components and the verifier agree on the election event context and in particular the list of ballot boxes and the number of voters that should be verified.

Verification 5.04 VerifySchnorrProofs

Context:
The Election Event Context ▷ Including $p, q, g, \mathtt{ee}, \psi_{\mathtt{max}}, \delta_{\mathtt{max}}$ ▷ See table 3 – Election Event Context

Input: ▷ All inputs are taken from table 3 – Setup Component Public Keys
CCR Choice Return Codes encryption keys $\left(\mathbf{pk}_{\mathrm{CCR}_1}, \mathbf{pk}_{\mathrm{CCR}_2}, \mathbf{pk}_{\mathrm{CCR}_3}, \mathbf{pk}_{\mathrm{CCR}_4}\right) \in \left(\mathbb{G}_q^{\psi_{\mathtt{max}}}\right)^4$
CCR Schnorr proofs of knowledge $\left(\boldsymbol{\pi}_{\mathsf{pkCCR},1}, \boldsymbol{\pi}_{\mathsf{pkCCR},2}, \boldsymbol{\pi}_{\mathsf{pkCCR},3}, \boldsymbol{\pi}_{\mathsf{pkCCR},4}\right) \in \left((\mathbb{Z}_q \times \mathbb{Z}_q)^{\psi_{\mathtt{max}}}\right)^4$
CCM election public keys $\left(\mathsf{EL}_{\mathsf{pk},1}, \mathsf{EL}_{\mathsf{pk},2}, \mathsf{EL}_{\mathsf{pk},3}, \mathsf{EL}_{\mathsf{pk},4}\right) \in \left(\mathbb{G}_q^{\delta_{\mathtt{max}}}\right)^4$
CCM Schnorr proofs of knowledge $\left(\boldsymbol{\pi}_{\mathsf{ELpk},1}, \boldsymbol{\pi}_{\mathsf{ELpk},2}, \boldsymbol{\pi}_{\mathsf{ELpk},3}, \boldsymbol{\pi}_{\mathsf{ELpk},4}\right) \in \left((\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\mathtt{max}}}\right)^4$
Electoral board public key $\mathsf{EB}_{\mathsf{pk}} \in \mathbb{G}_q^{\delta_{\mathtt{max}}}$
Electoral board Schnorr proofs of knowledge $\boldsymbol{\pi}_{\mathsf{EB}} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta_{\mathtt{max}}}$

Require: $\delta_{\mathtt{max}} - 1 \leq \psi_{\mathtt{max}}$

Operation:
```
1: Prepare the *Input* containing all public keys and Schnorr proofs
2: VerifSchnorrKeyGeneration ← VerifyKeyGenerationSchnorrProofs(*Input*)   ▷ see system specification
3: if VerifSchnorrKeyGeneration then
4:     return ⊤
5: else
6:     return ⊥
7: end if
```

Output:
The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.

~~Usually, each verification in the Swiss Post verifier performs only one specific check. The verification **??** deviates from this principle and checks the following:~~

- ~~the signatures by the setup component on the verification data~~
- ~~the signatures by the control components on the code shares~~
- ~~the control components correctly exponentiated the encrypted partial Choice Return Codes $\mathbf{pCC}$~~

- ~~the control components correctly exponentiated the encrypted Confirmation Key $\mathtt{CK}_{\mathtt{id}}$.~~

~~The verifier combines these checks into one larger verification for performance reasons, since the size of the data increases proportionally to the product of voting options and number of voters.~~
~~The Swiss Post Voting System splits the setup component verification data and control component code shares into chunks to keep the size manageable. The verifier checks that *all* signatures and *all* exponentiation proofs from *all* chunks of *all* verification card sets validate.~~
~~The verifier expects that every verification card set contains at least one verification card.~~
~~Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$ Vector of verification card set IDs $\mathbf{vcs} = (\mathtt{vcs}_0, \ldots, \mathtt{vcs}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Ba}$~~
~~The trust store containing the system's certificates~~
~~The message SetupComponentVerificationData from table 4 and its signature $\mathtt{s}_{\mathtt{SCVD}} \in \mathcal{B}^*$~~
~~The message ControlComponentCodeShares from table 4 and its signature $\mathtt{s}_{\mathtt{CCCS}} \in \mathcal{B}^*$~~
~~$\mathsf{SignatureSCVDVerif}_i \leftarrow \mathsf{VerifySignatureSetupComponentVerificationData}$ (SetupComponentVerificationData~~
~~$\mathsf{SignatureCCCSVerif}_{j,i} \leftarrow \mathsf{VerifySignatureControlComponentCodeShares}$ (ControlComponentCodeShares, $\mathtt{s}_{\mathtt{CC}}$~~
~~$\mathsf{vcsEncryptedPCCVerif}_{j,i} \leftarrow$ (Context and Input for verification card set $\mathtt{vcs_i}$ and control component j)~~
~~$\mathsf{vcsEncryptedCKVerif}_{j,i} \leftarrow$ (Context and Input for verification card set $\mathtt{vcs_i}$ and control component j)~~
~~$\top$ $\bot$~~
~~The result of the verification: $\top$ if all verifications are successful for *all* verification card sets, $\bot$ otherwise.~~

3.6 ~~Supporting Algorithms~~

~~The verifications in VerifyConfigPhase rely on the following algorithms.~~
~~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 $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$ Vector of verification card IDs $\mathbf{vc} = (\mathtt{vc}_0, \ldots, \mathtt{vc}_{\mathtt{N_E}-1}) \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}} \times \mathtt{N_E}}$ Number of voting options for this verification card set $n \in [1, \psi_{\mathsf{sup}}]$~~
~~Encrypted, hashed partial Choice Return Codes $\mathbf{c}_{\mathsf{pCC}} \in ({\mathbb{G}_q}^{n+1})^{\mathtt{N_E}}$ Voter Choice Return Code Generation public keys $\mathbf{K}_j \in {\mathbb{G}_q}^{\mathtt{N_E}}$ Exponentiated, encrypted, hashed partial Choice Return Codes $\mathbf{c}_{\mathsf{expPCC},j} \in ({\mathbb{G}_q}^{n+1})^{\mathtt{N_E}}$ Proofs of correct exponentiation $\boldsymbol{\pi}_{\mathsf{expPCC},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\mathtt{N_E}}$~~
~~$\mathbf{g} \leftarrow (g, \mathsf{c}_{\mathsf{pCC},\mathtt{id}})$ $\mathbf{y} \leftarrow (\mathsf{K}_{j,\mathtt{id}}, \mathsf{c}_{\mathsf{expPCC},j,\mathtt{id}})$ $\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathtt{ee}, \mathtt{vc}_{\mathtt{id}}, \text{"}\mathtt{GenEncLongCodeShares}\text{"}, \mathsf{IntegerToString}(j))$~~
~~$\mathsf{exponentiationVerif}_{id} \leftarrow \mathsf{VerifyExponentiation}(\mathbf{g}, \mathbf{y}, \pi_{\mathsf{expPCC},j,\mathtt{id}}, \mathbf{i}_{\mathsf{aux}})$ $\top$ $\bot$~~
~~The result of the verification: $\top$ if the verification is successful for *this specific* verification card set, $\bot$ otherwise.~~
~~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 $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$ Vector of verification card IDs $\mathbf{vc} = (\mathtt{vc}_0, \ldots, \mathtt{vc}_{\mathtt{N_E}-1}) \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}} \times \mathtt{N_E}}$~~
~~Encrypted, hashed Confirmation Key $\mathbf{c}_{\mathsf{ck}} \in ({\mathbb{G}_q}^{2})^{\mathtt{N_E}}$ Voter Vote Cast Return Code Generation public keys $\mathbf{Kc}_j \in {\mathbb{G}_q}^{\mathtt{N_E}}$ Exponentiated, encrypted, hashed Confirmation Key $\mathbf{c}_{\mathsf{expCK},j} \in ({\mathbb{G}_q}^{2})^{\mathtt{N_E}}$ Proofs of correct exponentiation $\boldsymbol{\pi}_{\mathsf{expCK},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\mathtt{N_E}}$~~
~~$\mathbf{g} \leftarrow (g, \mathsf{c}_{\mathsf{ck},\mathtt{id}})$ $\mathbf{y} \leftarrow (\mathsf{Kc}_{j,\mathtt{id}}, \mathsf{c}_{\mathsf{expCK},j,\mathtt{id}})$ $\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathtt{ee}, \mathtt{vc}_{\mathtt{id}}, \text{"}\mathtt{GenEncLongCodeShares}\text{"}, \mathsf{IntegerToString}(j))$~~
~~$\mathsf{exponentiationVerif}_{id} \leftarrow \mathsf{VerifyExponentiation}(\mathbf{g}, \mathbf{y}, \pi_{\mathsf{expCK},j,\mathtt{id}}, \mathbf{i}_{\mathsf{aux}})$ $\top$ $\bot$~~
~~The result of the verification: $\top$ if the verification is successful for *this specific* verification card set, $\bot$ otherwise.~~

4 Final Verification - VerifyTally

This section presents the verifications needed to ascertain the final tally reflects the choices made by the voters. Since this is performed after the voting period has closed, typically slightly over a month after the first verification run, the verifier verifies the tally phase using the context dataset obtained in the VerifyConfigPhase.
VerifyTally succeeds *only* if the validation of *all* ballot boxes succeed. Since every control component in the tally phase verifies the output of the preceding control components, failed verifications are unlikely to happen during VerifyTally. The most probable failure scenario would be a failed verification of the Tally control component's output—since no control component verifies these operations. However, in that case, one could easily re-run the Tally control component and re-verify the results.

4.1 Final - Completeness

VerifyTally requires both the context (table 3) and the tally dataset (table 6).
The placeholders represent the different identifiers present in the paths. To be more specific, the placeholder `${j}` represents the control component index and the placeholder `${bb}` represents the ballot box identifier. The different consistency verifications ensure that these indices and identifiers are consistent to the information in the ~~election event context~~Election Event Context.

Description	Path
Control Component Ballot Box	`tally/ballotBoxes/${bb}/c`
Online Control Component Shuffle	`tally/ballotBoxes/${bb}/c`
Tally Control Component Shuffle	`tally/ballotBoxes/${bb}/t`
Tally Control Component Votes	`tally/ballotBoxes/${bb}/t`
Tally Control Component ~~Decryptions `tally/evoting-decrypt.xml`~~ ~~Tally Control Component~~ Detailed Results	`tally/eCH-0222.xml`
~~Tally Control Component Results `tally/eCH-0110.xml`~~	

Tab. 6: The contents of the tally dataset. The tally dataset is used only in VerifyTally

Verification 6.01 VerifyTallyCompleteness

Input:
The context and tally datasets.

Operation:
1: ~~Check~~ Verify that the datasets contain all required elements from table 3 and table 6.

Output:
$\top$ if the verification succeeds, $\perp$ otherwise.

4.2 Final - Authenticity

Table 7 ~~lists the elements that must be signed and the required signers~~provides an overview of the authenticity checks for the tally verification. Each element corresponds to an entry in table 6, and provides the details of what should be given as input to the `VerifySignature` algorithm.

Message Name	Signer	Message Content	Context Data
ControlComponentBallotBox	Online CC$_j$	~~$(\{\mathtt{vc}_{j,i}, \mathtt{E1}_{j,i}, \widetilde{\mathtt{E1}}_{j,i}, \mathtt{E2}_{j,i}, \pi_{\mathtt{Exp},j,i}, \pi_{\mathtt{EqEnc},j,i}\}_{i=0}^{\mathtt{N_E}-1})$~~ ControlComponentBallotBoxPayload, see system specification table 18	("`ballotbox`", j, `ee`, `bb`)
ControlComponentShuffle	Online CC$_j$	~~$(\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \pi_{\mathsf{dec},j})$~~ ControlComponentShufflePayload, see system specification table 18	("`shuffle`", j, `ee`, `bb`)
TallyComponentShuffle	Tally CC	~~$(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \pi_{\mathsf{dec},5})$~~ TallyComponentShufflePayload, see system specification table 18	("`shuffle`", "`offline`", `ee`, `bb`)
TallyComponentVotes	Tally CC	~~$(L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$~~ TallyComponentVotesPayload, see system specification table 18	("`decoded votes`", `ee`, `bb`)
~~TallyComponentDecrypt~~ ~~Tally CC evoting decrypt XML ("evoting decrypt")~~ TallyComponentEch0222	Tally CC	~~eCH 0222 XML ("eCH 0222")~~ ~~TallyComponentEch0110 Tally CC~~ ~~eCH 0110 XML~~ eCH-0222 XML, see system specification table 17	~~("eCH 0110")~~ -

Tab. 7: Overview of the authenticity checks for the final verification

~~See section 3.2 for the signature of XML documents~~The eCH-0222 XML is signed with the algorithm GenXMLSignature as described in the section `Channel Security` of the system specification [11].

Verification 7.01 VerifySignatureControlComponentBallotBox

Context:
The trust store containing the system's certificates

Input:
The message ControlComponentBallotBox from table 7
The signature $s \in \mathcal{B}^*$

```
Operation: ~~$\mathsf{VerifySignature}(\text{“}\mathtt{control\_component\_j}\text{”}, (\{\mathtt{vc}_{j,i}, \mathtt{E1}_{j,i}, \widetilde{\mathtt{E1}}_{j,i}, \mathtt{E2}_{j,i}, \pi_{\mathtt{Exp},j,i}, \pi_{\mathtt{EqEnc},j,i}\}_{i=0}^{\mathtt{N_E}-1}), (\text{“}\mathtt{ballotbox}\text{”}, j, \mathtt{ee}, \mathtt{bb}), s)$~~ ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm
1: for $i \in [0, \mathtt{N_E})$ do
2:     $\mathtt{hVotes}_i \leftarrow \left((\mathtt{ee}, \mathtt{vcs}_i, \mathtt{vc}_{j,i}), \mathtt{E1}_{j,i}, \widetilde{\mathtt{E1}}_{j,i}, \mathtt{E2}_{j,i}, \pi_{\mathtt{Exp},j,i}, \pi_{\mathtt{EqEnc},j,i}\right)$
3: end for
4: $\mathtt{hVotes} \leftarrow (\mathtt{hVotes}_0, \ldots, \mathtt{hVotes}_{\mathtt{N_E}-1})$
5: $h \leftarrow \left((p, q, g), \mathtt{ee}, \mathtt{bb}, j, \mathtt{hVotes}\right)$
6: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
7: $\mathsf{VerifySignature}(\text{“}\mathtt{control\_component\_j}\text{”}, d, (\text{“}\mathtt{ballotbox}\text{”}, j, \mathtt{ee}, \mathtt{bb}), s)$
```

Output:
⊤ if the verification succeeds, ⊥ otherwise.

Test values for the verification 7.01 are provided in verify-signature-control-component-ballot-box.json.

Verification 7.02 VerifySignatureControlComponentShuffle

Context:
The trust store containing the system's certificates

Input:
The message ControlComponentShuffle from table 7
The signature $s \in \mathcal{B}^*$

```
Operation: ~~$\mathsf{VerifySignature}(\text{“}\mathtt{control\_component\_j}\text{”}, (\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j}), (\text{“}\mathtt{shuffle}\text{”}, j, \mathtt{ee}, \mathtt{bb}), s)$~~ ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm
1: $\mathtt{hShuffle} \leftarrow (\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j})$
2: $\mathtt{hDecryption} \leftarrow (\mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j})$
3: $h \leftarrow \left((p, q, g), \mathtt{ee}, \mathtt{bb}, \mathtt{hShuffle}, \mathtt{hDecryption}\right)$
4: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
5: $\mathsf{VerifySignature}(\text{“}\mathtt{control\_component\_j}\text{”}, d, (\text{“}\mathtt{shuffle}\text{”}, j, \mathtt{ee}, \mathtt{bb}), s)$
```

Output:
⊤ if the verification succeeds, ⊥ otherwise.

Test values for the verification 7.02 are provided in verify-signature-control-component-shuffle.json.

Verification 7.03 VerifySignatureTallyComponentShuffle

Context:
The trust store containing the system's certificates

Input:
The message TallyComponentShuffle from table 7
The signature $s \in \mathcal{B}^*$

Operation: ~~VerifySignature("sdm_tally", $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$, ("shuffle", "offline", ee, bb), s)~~ ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm

```
1: hShuffle ← $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5})$
2: hDecryption ← $(\mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$
3: $h$ ← (ee, bb, hShuffle, hDecryption)
4: $d$ ← Base64Encode(RecursiveHash($h$))
5: VerifySignature("sdm_tally", $d$, ("shuffle", "offline", ee, bb), $s$)
```

Output:
$\top$ if the verification succeeds, $\bot$ otherwise.

Test values for the verification 7.03 are provided in verify-signature-tally-component-shuffle.json.

Verification 7.04 VerifySignatureTallyComponentVotes

Context:
The trust store containing the system's certificates

Input:
The message TallyComponentVotes from table 7
The signature $s \in \mathcal{B}^*$

Operation: ~~$\mathsf{VerifySignature}(\text{“}\mathtt{sdm\_tally}\text{”}, (L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}}), (\text{“}\mathtt{decoded\ votes}\text{”}, \mathtt{ee}, \mathtt{bb}), s)$~~

~~⊤ if the verification succeeds, ⊥ otherwise.~~
~~The trust store containing the system's certificates~~
~~The message TallyComponentDecrypt from table 7 The signature $s \in \mathcal{B}^*$~~ ▷ For all algorithms see the crypto primitives specification ▷ We use nested structures in this algorithm

```
1: ~~$\mathsf{VerifySignature}(\text{“}\mathtt{sdm\_tally}\text{”}, \mathsf{evoting\ decrypt\ XML}, (\text{“}\mathtt{evoting\ decrypt}\text{”}), s)$~~
       $\mathtt{hVotes} \leftarrow \left((\hat{\mathrm{p}}_0, \dots, \hat{\mathrm{p}}_{\psi-1})_0, \dots, (\hat{\mathrm{p}}_0, \dots, \hat{\mathrm{p}}_{\psi-1})_{\mathrm{N_C}-1}\right)$
2: $\mathtt{hDecodedVotes} \leftarrow \left((\hat{\mathrm{v}}_0, \dots, \hat{\mathrm{v}}_{\psi-1})_0, \dots, (\hat{\mathrm{v}}_0, \dots, \hat{\mathrm{v}}_{\psi-1})_{\mathrm{N_C}-1}\right)$
3: $\mathtt{hWriteIns} \leftarrow \left((\hat{\mathrm{s}}_0, \dots, \hat{\mathrm{s}}_{k-1})_0, \dots, (\hat{\mathrm{s}}_0, \dots, \hat{\mathrm{s}}_{k-1})_{\mathrm{N_C}-1}\right)$
4: $h \leftarrow \left((p, q, g), \mathtt{ee}, \mathtt{bb}, \mathtt{hVotes}, \mathtt{hDecodedVotes}, \mathtt{hWriteIns}\right)$
5: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$
6: $\mathsf{VerifySignature}(\text{“}\mathtt{sdm\_tally}\text{”}, d, (\text{“}\mathtt{decoded\ votes}\text{”}, \mathtt{ee}, \mathtt{bb}), s)$
```

Output:
⊤ if the verification succeeds, ⊥ otherwise.

~~The trust store containing the system's certificates~~
~~The message TallyComponentEch0222 from table 7 The signature $s \in \mathcal{B}^*$~~
~~$\mathsf{VerifySignature}(\text{“}\mathtt{sdm\_tally}\text{”}, \mathsf{eCH\ 0222\ XML}, (\text{“}\mathtt{eCH\ 0222}\text{”}), s)$~~
Test values for the verification 7.04 are provided in
verify-signature-tally-component-votes.json.
~~⊤ if theverification succeeds, ⊥ otherwise.~~

Verification 7.05 VerifySignatureTallyComponentECH0222

Context:

The trust store containing the system's certificates

Input:

~~The message TallyComponentEch0110 from table 7 The signature $s \in \mathcal{B}^*$~~ Signed eCH-0222 XML from table 7

Operation:

```
1: ~~VerifySignature("sdm_tally", eCH 0110 XML, ("eCH 0110"), $s$)~~
       VerifyXMLSignature(eCH-0222 XML, $\mathsf{pk}_{\mathtt{tallyComponent}}$)   ▷ see system specification. The
   tally component's public key $\mathsf{pk}_{\mathtt{tallyComponent}}$ is extracted from the trust store.
```

Output:

$\top$ if the verification succeeds, $\perp$ otherwise.

4.3 Final - Consistency

For final - consistency we have the following verifications~~8.08-8.01~~.

Verification 8.01 VerifyEncryptionGroupConsistency

Input:

The ~~following files from the tally dataset in table 6: -~~ Election Event Context - ~~Control Component Ballot Box~~ see table 3.
The files from table 6.

Operation:

Verify that the ~~confirmed encrypted votes—including the verification card ID $\mathtt{vc_{id}}$, the encrypted vote $\mathtt{E1}$, the exponentiated encrypted vote $\widetilde{\mathtt{E1}}$, the encrypted partial Choice Return Codes $\mathtt{E2}$, and the zero-knowledge proofs $\pi_{\mathtt{Exp}}$ and $\pi_{\mathtt{EqEnc}}$—are identical across all control components—order notwithstanding~~encryption group parameters are consistent across all files.

Output:

$\top$ if ~~the confirmed encrypted votes are consistent~~all encryption group parameters are identical, $\bot$ otherwise.

Verification 8.02 VerifyNodeIdsConsistency

Input:

The ~~Election Event Context from table 3. The~~ node IDs included in the following files from ~~the tally dataset in~~ table 6:

- Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ▷ 1 per component ~~- Tally Control Component Shuffle~~

Operation:

1: ~~For each ballot box and every file, verify that the number of ciphertext elements equals the number allowed write-ins plus one~~ Verify that the node IDs are consistent across all files, i.e. that all files have exactly one contribution from each node.

Output:

$\top$ if the ~~ciphertexts have the expected number of elements in all ballot boxes~~node IDs are consistent, $\bot$ otherwise.

Verification 8.03 VerifyFileNameNodeIdsConsistency

Input:

The ~~Election Event Context from table 3.~~ following files from the tally dataset in table 6:

~~The Tally~~ - Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ~~from table 6.~~ ▷ 1 per component

Operation: ~~For each ballot box, verify that the number of plaintext elements after decryption equals the number of allowed write-ins plus one~~
Verify that the file names match the paths in the directory of the dataset.

Output:

⊤ if the ~~plaintexts have the expected number of elements in all ballot boxes~~file names are consistent, ⊥ otherwise.

Verification 8.04 VerifyElectionEventIdConsistency

Input:

The ~~verification card IDs from the setup component tally data from table 3. The~~ election event ID from the Election Event Context ~~from table 3to map the verification card set ID to the ballot box ID~~ - see table 3.
The ~~verification card IDs from the control component ballot box~~ election event ID included in the files from table 6.

Operation:

1: Verify that the ~~verification card IDs in the control component ballot boxes are a subset of the verification card IDs of the setup component tally data~~election event ID ee is consistent across all files.

Output:

⊤ if ~~all ballot boxes contain the expected verification card IDs~~the election event ID is consistent, ⊥ otherwise.

Verification 8.05 VerifyBallotBoxIdsConsistency

Input:

The ballot box IDs included in the files from table 6.

Operation:

1: Verify that the ballot box IDs are consistent across all files.
2: Verify that the path names containing the ballot box ID in the tally dataset match the ballot box ID within the files.

Output:

⊤ if the ballot box IDs are consistent, ⊥ otherwise.

Verification 8.06 VerifyFileNameBallotBoxIdsConsistency

Input:

The Election Event Context from table 3.
The paths of the tally dataset (table 6).

Operation:

1: Verify that path names of the tally dataset match the list of ballot box IDs in the Election Event Context.

Output:

$\top$ if the ballot box IDs between the tally dataset and the Election Event Context are consistent, $\bot$ otherwise.

Verification 8.07 VerifyVerificationCardIdsConsistency

Input:

The ~~following files from the tally dataset in table 3:~~ verification card IDs from the Setup Component Tally Data from table 3.
~~-~~ The Election Event Context from table 3 to map the verification card set ID to the ballot box ID.
The verification card IDs from the Control Component Ballot Box from table 6. ▷ 1 per component ~~- Online Control Component Shuffle - Tally Control Component Shuffle~~

Operation:

1: Verify that the ~~number of confirmed votes is identical in all files~~ verification card IDs in the Control Component Ballot Boxes are a subset of the verification card IDs of the Setup Component Tally Data.

Output:

$\top$ if ~~the number of confirmed votes is consistent~~all ballot boxes contain the expected verification card IDs, $\bot$ otherwise.

Verification 8.08 VerifyConfirmedEncryptedVotesConsistency

Input:

The ~~election event ID from the Election Event Context - see table 3.~~ following files from the tally dataset in table 6:
~~The election event ID included in the files from table 6.~~ - Control Component Ballot Box ▷ 1 per component

Operation:

1: Verify that the ~~election event ID $\mathtt{ee}$ is consistent across all files~~confirmed encrypted votes—including the verification card ID $\mathtt{vc_{id}}$, the encrypted vote $\mathtt{E1}$, the exponentiated encrypted vote $\widetilde{\mathtt{E1}}$, the encrypted partial Choice Return Codes $\mathtt{E2}$, and the zero-knowledge proofs $\pi_{\mathtt{Exp}}$ and $\pi_{\mathtt{EqEnc}}$—are identical across all control components—order notwithstanding.

Output:

$\top$ if the ~~election event ID is~~ confirmed encrypted votes are consistent, $\perp$ otherwise.

Verification 8.09 VerifyCiphertextsConsistency

Input:

The ~~node IDs included in the~~ Election Event Context from table 3.
The following files from the tally dataset in table 6:

- Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ▷ 1 per component
- Tally Control Component Shuffle

Operation:

1: ~~Verify that the node IDs are consistent across all files, i. e. that all files have exactly one contribution from each node~~For each ballot box and every file, verify that the number of ciphertext elements equals the number of allowed write-ins plus one.

Output:

$\top$ if the ciphertexts have the expected number of elements in all ballot boxes, $\perp$ otherwise.

Verification 8.10 VerifyPlaintextsConsistency

Input:

The Election Event Context from table 3.
The Tally Control Component Shuffle from table 6.

Operation:

1: For each ballot box, verify that the number of plaintext elements after decryption equals the number of allowed write-ins plus one.

Output:

⊤ if the ~~node IDs are consistent~~plaintexts have the expected number of elements in all ballot boxes, ⊥ otherwise.

Verification 8.11 VerifyNumberConfirmedEncryptedVotesConsistency

Input:

The following files from the tally dataset in table 6:
- Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ▷ 1 per component

~~Check that the file names match the paths in the directory of the datasets~~
~~⊤ if the file names are consistent, ⊥ otherwise .~~
~~The Election Event Context~~ - ~~see table 3.~~ Tally Control Component Shuffle
~~The files from table 6.~~ - Tally Control Component Votes

Operation:

1: Verify that the number of confirmed votes $\mathtt{N_C}$ is identical in Control Component Ballot Box and Tally Control Component Votes.
2: Verify that the number of mixed votes including trivial encryptions $\hat{\mathtt{N}}_\mathtt{C}$ is identical in Online Control Component Shuffle and Tally Control Component Shuffle.
3: Verify that if $\mathtt{N_C} < 2$ then $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C} + 2$, otherwise that $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C}$. ▷ If there are strictly less than 2 confirmed votes we add trivial encryptions to the ballot box. See algorithm GetMixnetInitialCiphertexts in the system specification

Output:

⊤ if ~~all encryption group parameters are identical~~the number of confirmed votes is consistent, ⊥ otherwise.

4.4 Final - Integrity

All integrity checks are performed as part of the domain definitions of the pseudocode for the verifications below.

4.5 Final - Evidence

After the tally phase, the verifier repeats the Tally control component's verification of the online control components and verifies the Tally control component's operations themselves. The verification of the online control component's verifications comprises the following:

- Verify the voting client's proofs in the algorithm VerifyVotingClientProofs (see system specification). This verification also ensures that the verifier works with the same primes mapping table pTable as the control components.
- Verify the online control components' shuffle and decryption proofs in the algorithm VerifyMixDecOffline (see system specification). To this end, the verifier invokes the GetMixnetInitialCiphertexts algorithm.

Moreover, the verifier must check the operations of the Tally control component:

- Verify the Tally control component's shuffle and decryption proofs.
- Verify the Tally control component's processing of the plaintexts.
- Verify the Tally control component's generation of the tally ~~files~~file.

Verification 10.01 VerifyOnlineControlComponents

Context:

Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$
Vector of verification card set IDs $\mathbf{vcs} = (\mathtt{vcs}_0, \dots, \mathtt{vcs}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{l_{ID}}})^{\mathtt{N_{bb}}}$
Vector of ballot box IDs $\mathbf{bb} = (\mathtt{bb}_0, \dots, \mathtt{bb}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{l_{ID}}})^{\mathtt{N_{bb}}}$
Election Event Context ▷ See table 3
Setup Component Public Keys ▷ See table 3

Input:

First Control Component Ballot Boxes $(\mathbf{vc}_1, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \boldsymbol{\pi}_{\mathtt{Exp},1}, \boldsymbol{\pi}_{\mathtt{EqEnc},1})$ for all $\mathtt{bb}_i$ ▷ See table 6
Online Control Component Shuffles $\{\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^{4}$ for all $\mathtt{bb}_i$ ▷ See table 6
Setup Component Tally Data $(\mathbf{vc}, \mathbf{K})$ for all $\mathtt{vcs}_i$ ▷ See table 3

Operation:

```
1: for $i \in [0, \mathtt{N_{bb}})$ do
2:     $\mathsf{Input}_{\mathtt{bb}_i} \leftarrow \left(\mathbf{vc}_1, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \boldsymbol{\pi}_{\mathtt{Exp},1}, \boldsymbol{\pi}_{\mathtt{EqEnc},1}, \{\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^{4}, \mathbf{vc}, \mathbf{K}\right)$
3:     $\mathsf{bbOnlineCCVerif}_i \leftarrow \mathsf{VerifyOnlineControlComponentsBallotBox}(\mathsf{Input}_{\mathtt{bb}_i})$    ▷ See Algorithm 4.1
4: end for
5: if $\mathsf{bbOnlineCCVerif}_i\ \forall\ i$ then
6:     return $\top$
7: else
8:     return $\bot$
9: end if
```

Output:

The result of the verification: $\top$ if the verification is successful for *all* ballot boxes, $\bot$ otherwise.

Algorithm 4.1 VerifyOnlineControlComponentsBallotBox

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 $\mathtt{ee} \in (\mathbb{A}_{Base16})^{1_{\mathtt{ID}}}$
Verification card set ID $\mathtt{vcs} \in (\mathbb{A}_{Base16})^{1_{\mathtt{ID}}}$
Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{1_{\mathtt{ID}}}$
Number of eligible voters $\mathtt{N_E} \in \mathbb{N}^+$ ▷ see system specification
Primes Mapping Table $\mathsf{pTable} \in \left(\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}{}^* \times \mathcal{T}_1^{50}\right)^n$ ▷ ψ and δ can be derived from pTable, see system specification
Election public key $\mathtt{EL_{pk}} = (\mathtt{EL_{pk,0}}, \ldots, \mathtt{EL_{pk,\delta_{max}-1}}) \in \mathbb{G}_q^{\delta_{\mathtt{max}}}$
CCM election public keys $(\mathtt{EL_{pk,1}}, \mathtt{EL_{pk,2}}, \mathtt{EL_{pk,3}}, \mathtt{EL_{pk,4}}) \in (\mathbb{G}_q^{\delta_{\mathtt{max}}})^4$
Electoral board public key $\mathtt{EB_{pk}} \in \mathbb{G}_q^{\delta_{\mathtt{max}}}$
Choice Return Codes encryption public key $\mathbf{pk}_{\mathrm{CCR}} \in \mathbb{G}_q^{\psi_{\mathtt{max}}}$

Input:

Control component's list of confirmed verification card IDs $\mathbf{vc}_1 = (\mathtt{vc}_{1,0}, \ldots, \mathtt{vc}_{1,\mathtt{N_C}-1}) \in \left((\mathbb{A}_{Base16})^{1_{\mathtt{ID}}}\right)^{\mathtt{N_C}}$
Control component's list of encrypted, confirmed votes $\mathbf{E1}_1 = (\mathtt{E1}_{1,0}, \ldots, \mathtt{E1}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q{}^{\delta+1}\right)^{\mathtt{N_C}}$
Control component's list of exponentiated, encrypted, confirmed votes $\widetilde{\mathbf{E1}}_1 = (\widetilde{\mathtt{E1}}_{1,0}, \ldots, \widetilde{\mathtt{E1}}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q{}^2\right)^{\mathtt{N_C}}$
Control component's list of encrypted, partial Choice Return Codes $\mathbf{E2}_1 = (\mathtt{E2}_{1,0}, \ldots, \mathtt{E2}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q{}^{\psi+1}\right)^{\mathtt{N_C}}$
Control component's list of exponentiation proofs $\boldsymbol{\pi}_{\mathsf{Exp},1} = (\pi_{\mathsf{Exp},1,0}, \ldots, \pi_{\mathsf{Exp},1,\mathtt{N_C}-1}) \in \left(\mathbb{Z}_q \times \mathbb{Z}_q\right)^{\mathtt{N_C}}$
Control component's list of plaintext equality proofs $\boldsymbol{\pi}_{\mathsf{EqEnc},1} = (\pi_{\mathsf{EqEnc},1,0}, \ldots, \pi_{\mathsf{EqEnc},1,\mathtt{N_C}-1}) \in \left(\mathbb{Z}_q \times \mathbb{Z}_q{}^2\right)^{\mathtt{N_C}}$
Preceding shuffled votes $\{\mathbf{c}_{\mathsf{mix},j}\}_{j=1}^4 \in \left((\mathbb{G}_q{}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$
Preceding shuffle proofs $\{\pi_{\mathsf{mix},j}\}_{j=1}^4$ ▷ See the domain of the shuffle argument in the crypto primitives specification
Preceding partially decrypted votes $\{\mathbf{c}_{\mathsf{dec},j}\}_{j=1}^4 \in \left((\mathbb{G}_q{}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$
Preceding decryption proofs $\{\boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^4 \in \left((\mathbb{Z}_q{}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$
Vector of verification card IDs $\mathbf{vc} = (\mathtt{vc}_0, \ldots, \mathtt{vc}_{\mathtt{N_E}-1}) \in ((\mathbb{A}_{Base16})^{1_{\mathtt{ID}}})^{\mathtt{N_E}}$
Verification card public keys $\mathbf{K} = (\mathtt{K}_0, \ldots, \mathtt{K}_{\mathtt{N_E}-1}) \in (\mathbb{G}_q)^{\mathtt{N_E}}$

Require: $\mathtt{N_E} \geq \mathtt{N_C}$
Require: $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C}$ if $\mathtt{N_C} \geq 2$, otherwise $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C} + 2$ ▷ The algorithm runs with at least two votes
Require: $\mathtt{vc}_{1,i} \neq \mathtt{vc}_{1,k}, \forall i, k \in \{0, \ldots, (\mathtt{N_C} - 1)\} \wedge i \neq k$ ▷ All verification card IDs must be distinct

Operation:

```
1: if $\mathtt{N_C} \geq 1$ then    ▷ Verifying the voting client proofs requires at least one confirmed vote
2:     $\mathbf{KMap} \leftarrow \left((\mathtt{vc}_0, \mathtt{K}_0), \ldots, (\mathtt{vc}_{\mathtt{N_E}-1}, \mathtt{K}_{\mathtt{N_E}-1})\right)$
3:     $\mathsf{vcProofsVerif} \leftarrow \mathsf{VerifyVotingClientProofs}\left(\mathbf{vc}_1, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \boldsymbol{\pi}_{\mathsf{Exp},1}, \boldsymbol{\pi}_{\mathsf{EqEnc},1}, \mathbf{KMap}\right)$    ▷ see system specification
4: else
5:     $\mathsf{vcProofsVerif} \leftarrow \top$
6: end if
7: $\mathbf{vcMap}_1 \leftarrow \left((\mathtt{vc}_{1,0}, \mathtt{E1}_{1,0}) \ldots, (\mathtt{vc}_{1,\mathtt{N_C}-1}, \mathtt{E1}_{1,\mathtt{N_C}-1})\right)$
8: $\mathbf{c}_{\mathsf{init},1} \leftarrow \mathsf{GetMixnetInitialCiphertexts}(\mathbf{vcMap}_1)$    ▷ see system specification
9: $\mathsf{shuffleProofsVerif} \leftarrow \mathsf{VerifyMixDecOffline}\left(\mathbf{c}_{\mathsf{init},1}, \{\mathbf{c}_{\mathsf{mix},j}\}_{j=1}^4, \{\pi_{\mathsf{mix},j}\}_{j=1}^4, \{\mathbf{c}_{\mathsf{dec},j}\}_{j=1}^4, \{\boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^4\right)$    ▷ see system specification
10: if $\mathsf{vcProofsVerif} \wedge \mathsf{shuffleProofsVerif}$ then
11:     return $\top$
12: else
13:     return $\perp$
14: end if
```

Output:

The result of the verification: $\top$ if the verification is successful for *this specific* ballot box, $\perp$ otherwise.

Next, the verifier checks the Tally control component's operations.

Verification 10.02 VerifyTallyControlComponent

Context:

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

Vector of ballot box IDs $\mathbf{bb} = (\mathtt{bb}_0, \ldots, \mathtt{bb}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{l_{ID}}})^{\mathtt{N_{bb}}}$

Election Event Context ▷ See table 3

Setup Component Public Keys ▷ See table 3

Input:

Last Online Control Component Shuffles $(\mathbf{c}_{\mathsf{mix},4}, \pi_{\mathsf{mix},4}, \mathbf{c}_{\mathsf{dec},4}, \boldsymbol{\pi}_{\mathsf{dec},4})$ for all $\mathtt{bb}_i$ ▷ See table 6

Tally Control Component Shuffles $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$ for all $\mathtt{bb}_i$ ▷ See table 6

Tally Control Component Votes $(L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$ for all $\mathtt{bb}_i$ ▷ See table 6

~~For each ballot box, the list of all selected decoded voting options~~

~~$L_{\mathsf{decodedVotes\,bb}} \in \left((\mathcal{T}_1^{50})^{\psi}\right)^{\mathtt{N_{Cbb}}}$~~ Election Event Configuration configuration XML ▷ See table 3

Tally Control Component ~~Decryptions evoting-decrypt XML~~ Detailed Results eCH-0222 XML ▷ See table 6

~~Tally Control Component Results eCH-0110 XML~~ Key-value map of $L_{\mathsf{decodedVotes}}$ per authorization name $\mathsf{Map}_{\mathtt{decodedVotes}}$

~~Tally Control Component Detailed Results eCH-0222 XML~~ Key-value map of L_{writeIns} per authorization name $\mathsf{Map}_{\mathtt{writeIns}}$

Operation:

```
1: for $i \in [0, \mathtt{N_{bb}})$ do
2:     $\mathsf{Input}_{\mathtt{bb}_i} \leftarrow (\mathbf{c}_{\mathsf{dec},4}, \mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5}, L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$
3:     $\mathsf{tallyVerif}_i \leftarrow \mathsf{VerifyTallyControlComponentBallotBox}(\mathsf{Input}_{\mathtt{bb}_i})$    ▷ See Algorithm 4.2
4: end for
5: ~~$\mathsf{Input}_{\mathsf{tallyFiles}} \leftarrow$ (configuration XML, evoting-decrypt XML, eCH-0110 XML, eCH-0222 XML, $L_{\mathsf{decodedVotes\,bb}}$)~~ $\mathsf{Input}_{\mathsf{eCH0222}} \leftarrow$ (configuration XML, eCH-0222 XML, $\mathsf{Map}_{\mathtt{decodedVotes}}$, $\mathsf{Map}_{\mathtt{writeIns}}$)
6: ~~$\mathsf{tallyFilesVerif} \leftarrow (\mathsf{Input}_{\mathsf{tallyFiles}})$~~ $\mathsf{eCH0222Verif} \leftarrow \mathsf{VerifyECH0222}(\mathsf{Input}_{\mathsf{eCH0222}})$    ▷ See Algorithm 4.4
7: if $\mathsf{tallyVerif}_i\ \forall\ i \wedge \mathsf{eCH0222Verif}$ then
8:     return $\top$
9: else
10:    return $\bot$
11: end if
```

Output:

The result of the verification: $\top$ if the verification is successful for *all* ballot boxes, $\bot$ otherwise.

Algorithm 4.2 VerifyTallyControlComponentBallotBox

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 $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$
Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$
Number of eligible voters $\mathtt{N_E} \in \mathbb{N}^+$ ▷ see system specification
Primes Mapping Table $\mathsf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times {\mathbb{A}_{UCS}}^* \times \mathcal{T}_1^{50})^n$ ▷ ψ and δ can be derived from pTable, see system specification
Electoral board public key $\mathtt{EB_{pk}} \in \mathbb{G}_q^{\delta_{\max}}$

Input:

The last online control component's partially decrypted votes $\mathbf{c}_{\mathsf{dec},4} \in ({\mathbb{G}_q}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}$
The tally component's shuffled votes $\mathbf{c}_{\mathsf{mix},5} \in ({\mathbb{G}_q}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}$
The tally component's shuffle proofs $\pi_{\mathsf{mix},5}$ ▷ See the domain of the shuffle argument in the crypto primitives specification
The decrypted votes $\mathbf{m} = (m_0, \dots, m_{\hat{\mathtt{N}}_\mathtt{C}-1}) \in ({\mathbb{G}_q}^{\delta})^{\hat{\mathtt{N}}_\mathtt{C}}$
The decryption proofs $\boldsymbol{\pi}_{\mathsf{dec},5} \in ({\mathbb{Z}_q}^{\delta+1})^{\hat{\mathtt{N}}_\mathtt{C}}$
List of ~~all selected encoded voting options~~ decrypted votes $L_{\mathsf{votes}} = (\hat{\mathbf{p}}_0, \dots, \hat{\mathbf{p}}_{\mathtt{N}_\mathtt{C}-1}) \in \left(((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^{\psi}\right)^{\mathtt{N}_\mathtt{C}}$
List of ~~all selected decoded voting options~~ decoded votes $L_{\mathsf{decodedVotes}} = (\hat{\mathbf{v}}_0, \dots, \hat{\mathbf{v}}_{\mathtt{N}_\mathtt{C}-1}) \in \left((\mathcal{T}_1^{50})^{\psi}\right)^{\mathtt{N}_\mathtt{C}}$
List of ~~all selected decoded write-in votes~~ decoded write-ins $L_{\mathsf{writeIns}} = (\hat{\mathbf{s}}_0, \dots, \hat{\mathbf{s}}_{\mathtt{N}_\mathtt{C}-1}) \in \left(({\mathbb{A}_{\mathsf{latin}}}^*)^*\right)^{\mathtt{N}_\mathtt{C}}$

Require: $\mathtt{N_E} \geq \mathtt{N_C}$
Require: $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C}$ if $\mathtt{N_C} \geq 2$, otherwise $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N_C} + 2$ ▷ The algorithm runs with at least two votes
Require: $\hat{\mathbf{p}}_i \subseteq \mathsf{GetEncodedVotingOptions}(()), \forall i \in \{0, \dots, (\mathtt{N_C} - 1)\}$
Require: $\hat{\mathrm{p}}_{i,k} \neq \hat{\mathrm{p}}_{i,l}, \forall i \in \{0, \dots, (\mathtt{N_C} - 1)\}, \forall k, l \in \{0, \dots, (\psi - 1)\} \land k \neq l$ ▷ A vote's selected encoded voting options must be distinct

Operation:

```
1:  $\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathtt{ee}, \mathtt{bb}, \text{“}\mathtt{MixDecOffline}\text{”})$
2:  $\mathtt{EB_{pk,cut}} \leftarrow (\mathtt{EB_{pk,0}}, \cdots, \mathtt{EB_{pk,\delta-1}})$
3:  $\mathsf{shuffleVerif} \leftarrow \mathsf{VerifyShuffle}(\mathbf{c}_{\mathsf{dec},4}, \mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathtt{EB_{pk,cut}})$    ▷ See crypto primitives specification
4:  for $i \in [0, \hat{\mathtt{N}}_\mathtt{C})$ do
5:      ~~$\mathsf{decryptVerif} \leftarrow \mathsf{VerifyDecryptions}(\mathbf{c}_{\mathsf{mix},5}, \mathtt{EB_{pk,cut}}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5}, \mathbf{i}_{\mathsf{aux}})$~~
        $\mathsf{decryptVerif}_i \leftarrow \mathsf{VerifyDecryption}(\mathbf{c}_{\mathsf{mix},5,i}, \mathtt{EB_{pk,cut}}, m_i, \boldsymbol{\pi}_{\mathsf{dec},5,i}, \mathbf{i}_{\mathsf{aux}})$
6:  end for                                         ▷ See crypto primitives specification
7:  $\mathsf{processVerif} \leftarrow \mathsf{VerifyProcessPlaintexts}(\mathbf{m}, L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$    ▷ See algorithm 4.3
8:  if $\mathsf{shuffleVerif} \land \mathsf{decryptVerif}_i \forall i \in [0, \hat{\mathtt{N}}_\mathtt{C}) \land \mathsf{processVerif}$ then
9:      return $\top$
10: else
11:     return $\bot$
12: end if
```

Output:

The result of the verification: $\top$ if the verification is successful for *this specific* ballot box, $\bot$ otherwise.

Algorithm 4.3 VerifyProcessPlaintexts

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 $\mathsf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times \mathbb{A}_{UCS}{}^* \times \mathcal{T}_1^{50})^n$ ▷ pTable is of the form $((\mathrm{v}_0, \tilde{\mathrm{p}}_0, \sigma_0, \tau_0), \dots, (\mathrm{v}_{n-1}, \tilde{\mathrm{p}}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$ ▷ ψ and δ can be derived from pTable, see system specification

Input: ▷ see system specification

List of plaintext votes $\mathbf{m} = (m_0, \dots m_{\hat{\mathtt{N}}_\mathtt{C}-1}) \in (\mathbb{G}_q{}^\delta)^{\hat{\mathtt{N}}_\mathtt{C}}$
List of ~~all selected encoded voting options~~ decrypted votes $L_{\mathsf{votes}} = (\hat{\mathbf{p}}_0, \dots, \hat{\mathbf{p}}_{\mathtt{N}_\mathtt{C}-1}) \in \left(((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^\psi\right)^{\mathtt{N}_\mathtt{C}}$ ▷ We assume that the algorithm 4.2 verified that all votes' selected encoded voting options are distinct and a subset of the possible encoded voting options.
List of ~~all selected decoded voting options~~ decoded votes $L_{\mathsf{decodedVotes}} = (\hat{\mathbf{v}}_0, \dots, \hat{\mathbf{v}}_{\mathtt{N}_\mathtt{C}-1}) \in \left((\mathcal{T}_1^{50})^\psi\right)^{\mathtt{N}_\mathtt{C}}$
List of ~~all selected decoded write-in votes~~ decoded write-ins $L_{\mathsf{writeIns}} = (\hat{\mathbf{s}}_0, \dots, \hat{\mathbf{s}}_{\mathtt{N}_\mathtt{C}-1}) \in \left((\mathbb{A}_{\mathsf{latin}}{}^*)^*\right)^{\mathtt{N}_\mathtt{C}}$

Require: $\hat{\mathtt{N}}_\mathtt{C} \geq 2$ ▷ The algorithm runs with at least two votes
Require: $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N}_\mathtt{C}$ if $\mathtt{N}_\mathtt{C} \geq 2$, otherwise $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N}_\mathtt{C} + 2$

Operation:

```
1: $\left(L'_{\mathsf{votes}}, L'_{\mathsf{decodedVotes}}, L'_{\mathsf{writeIns}}\right) \leftarrow \mathsf{ProcessPlaintexts}(\mathbf{m})$   ▷ see system specification
2: if $\left(L'_{\mathsf{votes}} = L_{\mathsf{votes}}\right) \wedge \left(L'_{\mathsf{decodedVotes}} = L_{\mathsf{decodedVotes}}\right) \wedge \left(L'_{\mathsf{writeIns}} = L_{\mathsf{writeIns}}\right)$ then
3:     return $\top$
4: else
5:     return $\bot$
6: end if
```

Output:

The result of the verification: $\top$ if the verification is successful for *this specific* ballot box, $\bot$ otherwise.

In the following algorithm, the verifier regenerates the eCH-0222 XML and verifies its correctness. Due to the complexity of the XML file structure, we utilize GenXMLSignature to ensure that the tally control component and the verifiers eCH-0222 XML are semantically identical. GenXMLSignature canonicalizes and transforms the XML document, eliminating potential ambiguities such as OS-dependent line breaks and inconsistent whitespace. To achieve this, we generate a temporary secret key for signing, which is exclusively used within the VerifyECH0222 algorithm. We assume a function GenSecretKey that generates a private signing key.

Algorithm 4.4 VerifyECH0222

Context:

Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{l_{\mathtt{ID}}}$ ~~Number of allowed selections for each ballot box $\psi \in [1, \psi_{\mathsf{sup}}]$~~

Input:

~~Configuration file~~ Election Event Configuration configuration XML ▷ See table 3
~~File aggregating the submitted votes by ballot box id evoting-decrypt XML File containing the tallied votes eCH-0110 XML~~ Tally Control Component Detailed Results eCH-0222 XML ▷ See table 6
~~File containing the detailed results eCH-0222 XML~~ Key-value map of $L_{\mathsf{decodedVotes}}$ per authorization name $\mathsf{Map}_{\mathtt{decodedVotes}}$
~~For each ballot box, the list of all selected decoded voting options $L_{\mathsf{decodedVotes bb}} \in \left((\mathcal{T}_1^{50})^{\psi}\right)^{\mathtt{N_{Cbb}}}$~~ Key-value map of L_{writeIns} per authorization name $\mathsf{Map}_{\mathtt{writeIns}}$

Operation:

```
1: ~~evoting-decrypt XML′ ← Aggregate the decoded voting options from all ballot boxes.~~
   eCH-0222 XML' ← CreateECH0222(configuration XML, $\mathsf{Map}_{\mathtt{decodedVotes}}$, $\mathsf{Map}_{\mathtt{writeIns}}$)   ▷ see system specification
2: ~~eCH-0110 XML′ ← Count how many votes each voting option received using ee, in the format defined under . Moreover, the eCH-0110 XML file must declare votes for party lists without a candidate selection as invalid votes (if the election imposes this rule), as explained in the system specification [11], in the section Valid Combinations of Voting Options.~~ sk ← GenSecretKey()   ▷ Create a temporary secret key for signing
3: ~~eCH-0222 XML′ ← Establish the detailed results containing the raw decrypted votes using ee, in the format defined under .~~ $\mathrm{D_{signed}}$ ← GenXMLSignature(eCH-0222 XML, sk)
   ▷ see system specification
4: $\mathrm{D'_{signed}}$ ← GenXMLSignature(eCH-0222 XML', sk))
5: $\mathrm{d}$ ← ExtractDigest($\mathrm{D_{signed}}$)   ▷ see system specification
6: $\mathrm{d'}$ ← ExtractDigest($\mathrm{D'_{signed}}$)
7: if $\mathrm{d} = \mathrm{d'}$ then
8:     return ~~evoting-decrypt XML = evoting-decrypt XML′ ∧ eCH-0110 XML = eCH-0110 XML′ ∧ eCH-02~~
       $\top$
9: else
10:     return $\bot$
11: end if
```

Output:

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

As noted in the system specification ~~[11]~~, the details of the rules taken into account for the constitution of the XML ~~files~~ file are beyond the scope of this document, but the ~~schemas are attached or referenced~~ schema is attached there.

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:

- 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 (Université catholique Louvain)
- Vanessa Teague (Thinking Cybersecurity)

References

[1] Die Schweizerische Bundeskanzlei (BK): *Federal Chancellery Ordinance on Electronic Voting (OEV), 01 July 2022.*

[2] 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.*

[3] 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.

[4] gfs.bern: *Vorsichtige Offenheit im Bereich digitale Partizipation - Schlussbericht.* Mar. 2020.

[5] R. Haenni, E. Dubuis, R. E. Koenig, and P. Locher: "CHVote: Sixteen Best Practices and Lessons Learned". In: *International Joint Conference on Electronic Voting.* Springer. 2020, pp. 95–111.

[6] R. Haenni, E. Dubuis, R. E. Koenig, and P. Locher: "Process models for universally verifiable elections". In: *International Joint Conference on Electronic Voting.* Springer. 2018, pp. 84–99.

[7] S. Josefsson et al.: *The base16, base32, and base64 data encodings.* Tech. rep. RFC 4648, October, 2006.

[8] B. Smyth: "A foundation for secret, verifiable elections." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 225.

[9] 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.

[10] 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.

[11] Swiss Post: *Swiss Post Voting System. System Specification. Version 1.5.0.* https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/System. 2025.

List of Algorithms

List of Verifications

List of Figures

List of Tables