Swiss Post Voting System

Verifier specification

Swiss Post*

Version 1.3.1

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.

Revision chart

Version	Description	Author	Reviewer	Date
0.9	First published version	OE	XM, TH	2021-09-01
0.9.1	See change log for version 0.9.1	OE, HR	XM, JS, TH	2021-10-15
0.9.2	See change log for version 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	TH, OE	XM, JS, HR	2022-08-19
1.1.0	See change log	TH, OE	XM, JS, HR	2022-10-03
1.2.0	See change log	TH, OE	XM, JS, HR	2022-10-31
1.3.0	See change log	AH, OE	XM, JS, TH	2022-12-09
1.3.1	See change log	AH, OE	XM, JS, TH	2023-02-23

Contents

Symbols

$\mathbb{A}_{Base16}$	Base16 (Hex) alphabet [7]
$\mathbb{A}_{UCS}$	Alphabet of the Universal Coded Character Set (UCS) according to ISO/IEC10646
$\mathcal{T}_1^{50}$	Alphabet used for voting option identifiers (word characters, minus sign and underscore sign: `[\w\-_]{1,50}`)
$\mathcal{B}^*$	Set of byte arrays of arbitrary length
δ	Number of elements of the election public key
$\hat{\delta}$	Number of write-in options + 1 for a specific verification card set
δ	Maximum of $\hat{\delta}$ over all verification card sets
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 $\mathbb{Q}_p$
$\mathbb{H}_l$	Ciphertext domain ($= \underbrace{\mathbb{G}_q \times \cdots \times \mathbb{G}_q}_{l+1 \text{ times}}$)
$\mathtt{L_{ID}}$	Character length of unique identifiers
L_{votes}	List of decrypted, processed votes
$L_{\mathsf{decodedVotes}}$	List of decoded votes
L_{writeIns}	List of decoded write-in votes
μ	Maximum supported number of write-in options + 1
$\mathbf{m}$	List of plaintext votes
φ	Number of elements of the Choice Return Codes Encryption public key
$\mathbb{N}^*$	Set of strictly positive integer numbers
$\mathtt{N_C}$	Number of confirmed votes
$\hat{\mathtt{N}}_\mathtt{C}$	Number of mixed votes including trivial encryptions
$\mathtt{N_E}$	Number of eligible voters of a specific verification card set
n	Number of voting options of a specific verification card set
n_{total}	Number of distinct voting options across all verification card sets
ω	Maximum supported number of voting options
φ	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 $(\tilde{\mathrm{p}}_1, \ldots, \tilde{\mathrm{p}}_n),\ \tilde{\mathrm{p}}_k \in (\mathbb{G}_q \cap \mathbb{P}) \setminus g$
$\tilde{\mathbf{v}}$	Voting options $(\mathrm{v}_0, \ldots, \mathrm{v}_{n-1})$
p	Encryption group modulus
q	Encryption group cardinality s.t. $p = 2q + 1$
$\lvert x \rvert$	Bit length of the number x
$\mathbf{w}_\mathtt{id}$	Voter's encoded write-ins $(\mathrm{w}_{\mathtt{id},0}, \ldots, \mathrm{w}_{\mathtt{id},\hat{\delta}-2})$
$\mathbb{Z}_q$	Set of integers modulo q

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 [9]:

- 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 5.3.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 2.4.

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

We use the following conventions throughout the document.

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

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

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

  results in a list without any nested structures:

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

- Use the union operator $\cup$ to indicate that we append an element to a list:

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

  $$list \leftarrow list \cup c$$

  results in

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

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

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

  $$a \in list = \top$$

  $$c \in list = \bot$$

- Define sets for ranges, e.g. for $i \in [0, n)$. This expression indicates that we include the lower bound but exclude the upper bound, i.e. $0 \leq i < n$;

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 [8].

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 pseudo-code 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 pseudo-code 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 pseudo-code specifications, 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 [8], 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` algorithm from the crypto primitives specification. 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 specifications, 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 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.

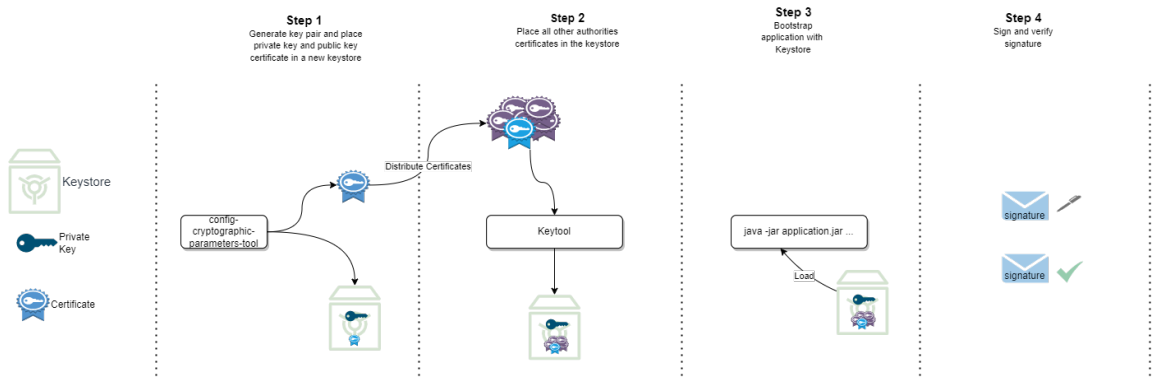

Fig. 1: The control components generate a certificate, share it out of band with the auditors, who import them in their keystore, thus enabling them to directly verify the authenticity of signed messages.

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

2.4 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 data set being used for the audit by printing the hash value of the audit archive.

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:

- Name and date of the election event
- Number of elections and votes
- Number of productive and test ballot boxes
- Number of real and test voters

The auditors can learn this information 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 configuration of the election event—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 pseudo-code indicates the checks that the auditors must perform manually.

Verification 0.01 ManualChecksByAuditors

Context:

A verifier's execution in the VerifyConfigPhase or VerifyTally.

Input:

The verifier's report indicating the domain-specific parameters.
The protocol participants' certificates—received via an out-of-band channel (section 2.3).

Operation:

1: Check that the certificates' fingerprints match the expected ones.
2: Check that the name and date of the election event correspond to the expected ones.
3: Check that the number of elections and votes corresponds to the expected one.
4: Check that the number of productive and test ballot boxes corresponds to the expected one.
5: Check that the number of real and test voters corresponds to the expected one.
6: Check that all expected verifications executed successfully.

▷ See crypto primitives specification

Output:

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

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, the auditors should record some of the election results, which can be compared against the official published results at a later time. This record-keeping can help to identify any potential discrepancies and 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.

2.5 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.

2.6 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 required elements are provided in table 2.

Description	Path
Encryption Parameters	`setup/encryptionParametersPayload.json`
Election Event Configuration	`setup/configuration-anonymized.xml`
Setup Component Public Keys	`setup/setupComponentPublicKeys.json`
Election Event Context	`setup/electionEventContextPayload.json`
Online Control Component Public Keys	`setup/controlComponentPublicKeysPayload.${j}.json`
Setup Component Verification Data	`setup/verification_card_sets/${vcs}/setupComponentVerificationDataPayload.${chunkId}.json`
Control Component Code Shares	`setup/verification_card_sets/${vcs}/controlComponentCodeSharesPayload.${chunkId}.json`
Setup Component Tally Data	`setup/verification_card_sets/${vcs}/setupComponentTallyDataPayload.json`

Tab. 2: The required elements for the setup verification, along with their path within the audit archive

3.2 Setup - Authenticity

Table 3 provides an overview of the authenticity checks for the setup verification. Each element corresponds to an entry in table 2, and provides the details of what should be given as input to the `VerifySignature` algorithm.

Message Name	Signer	Message Content	Context Data
SetupComponentEncryptionParameters	Setup Comp.	$(p, q, g, \texttt{seed}, \mathbf{p})$	("`encryption parameters`")
CantonConfig	Canton	configuration XML	("`configuration`")
SetupComponentPublicKeys	Setup Comp.	$(\{j, \mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathtt{pkCCR},j}, \mathtt{EL}_{\mathtt{pk},j}, \pi_{\mathtt{ELpk},j}\}_{j=1}^{4}, \mathtt{EB}_{\mathtt{pk}}, \pi_{\mathtt{EB}}, \mathtt{EL}_{\mathtt{pk}}, \mathbf{pk}_{\mathrm{CCR}})$	("`public keys`", "`setup`", `ee`)
ElectionEventContext	Setup Comp.	Configuration of the Election Event: See System Specification section 3.3 and 3.4	("`election event context`", `ee`)
ControlComponentPublicKeys	CC_j	$(\mathbf{pk}_{\mathrm{CCR}_j}, \mathtt{EL}_{\mathtt{pk},j}, \pi_{\mathtt{ELpk},j})$	("`OnlineCC keys`", j, `ee`)
SetupComponentVerificationData	Setup Comp.	$(\{\mathtt{vc}_{\mathtt{id}}, \mathtt{K}_{\mathtt{id}}, \mathtt{c}_{\mathtt{pCC},\mathtt{id}}, \mathtt{c}_{\mathtt{ck},\mathtt{id}}\}_{id=0}^{\mathtt{N_E}-1}, \mathtt{L}_{\mathtt{pCC}})$	("`verification data`", `ee`, `vcs`)
ControlComponentCodeShares	CC_j	$(\{\mathtt{vc}_{\mathtt{id}}, \mathtt{K}_{j,id}, \mathtt{Kc}_{j,id}, \mathtt{c}_{\mathtt{expPCC},j,id}, \mathtt{c}_{\mathtt{expCK},j,id}, \pi_{\mathtt{expPCC},j,id}, \pi_{\mathtt{expCK},j,id}\}_{id=0}^{\mathtt{N_E}-1})$	("`encrypted code shares`", j, `ee`, `vcs`)
SetupComponentTallyData	Setup Comp.	$(\mathbf{vc}, \mathbf{K})$	("`tally data`", `ee`, `vcs`)

Tab. 3: Overview of the authenticity checks for the setup verification

The configuration XML above, as well as the two other XML files mentioned in table 5, 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 `"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, ...) are represented as strings,
- the signature field itself must be ignored.

Verification 2.01 VerifySignatureSetupComponentEncryptionParameters

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentEncryptionParameters from table 3
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("`sdm_config`", $(p, q, g, seed, \mathbf{p})$, ("`encryption parameters`"), s)
▷ See crypto primitives specification

Output:

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

Verification 2.02 VerifySignatureCantonConfig

Context:

The trust store containing the system's certificates

Input:

The message CantonConfig from table 3

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

Operation:

1: VerifySignature("`canton`", configuration XML, ("`configuration`"), s)
▷ See crypto primitives specification

Output:

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

Verification 2.03 VerifySignatureSetupComponentPublicKeys

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentPublicKeys from table 3

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

Operation:

1: VerifySignature("`sdm_config`",
$(\{j, \mathbf{pk}_{\mathrm{CCR}_j}, \boldsymbol{\pi}_{\mathsf{pkCCR},j}, \mathrm{EL}_{\mathtt{pk},j}, \boldsymbol{\pi}_{\mathsf{ELpk},j}\}_{j=1}^{4}, \mathrm{EB}_{\mathtt{pk}}, \boldsymbol{\pi}_{\mathsf{EB}}, \mathrm{EL}_{\mathtt{pk}}, \mathbf{pk}_{\mathrm{CCR}})$,
("`public keys`", "`setup`", `ee`), s)
▷ See crypto primitives specification

Output:

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

Verification 2.04 VerifySignatureControlComponentPublicKeys

Context:

The trust store containing the system's certificates

Input:

The message ControlComponentPublicKeys from table 3

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

Operation:

1: VerifySignature("control_component_j",
$(\mathbf{pk}_{\mathrm{CCR}_j}, \mathtt{EL}_{\mathtt{pk},j}, \boldsymbol{\pi}_{\mathtt{ELpk},j})$,
("OnlineCC keys", j, ee), s)

▷ See crypto primitives specification

Output:

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

Verification 2.05 VerifySignatureSetupComponentVerificationData

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentVerificationData from table 3

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

Operation:

1: VerifySignature("sdm_config",
$(\{\mathtt{vc}_{\mathtt{id}}, \mathtt{K}_{\mathtt{id}}, \mathtt{c}_{\mathtt{pCC},\mathtt{id}}, \mathtt{c}_{\mathtt{ck},\mathtt{id}}\}_{id=0}^{\mathtt{N_E}-1}, \mathtt{L}_{\mathtt{pCC}})$,
("verification data", ee, vcs), s)

▷ See crypto primitives specification

Output:

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

Verification 2.06 VerifySignatureControlComponentCodeShares

Context:

The trust store containing the system's certificates

Input:

The message ControlComponentCodeShares from table 3
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("control_component_j", $(\{\mathtt{vc}_{\mathtt{id}}, \mathtt{K}_{j,id}, \mathtt{Kc}_{j,id}, \mathtt{c}_{\mathtt{expPCC},j,id}, \mathtt{c}_{\mathtt{expCK},j,id}, \pi_{\mathtt{expPCC},j,id}, \pi_{\mathtt{expCK},j,id}\}_{id=0}^{\mathtt{N_E}-1})$, ("encrypted code shares", j, ee, vcs), s)
▷ See crypto primitives specification

Output:

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

Verification 2.07 VerifySignatureSetupComponentTallyData

Context:

The trust store containing the system's certificates

Input:

The message SetupComponentTallyData from table 3
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("sdm_config", $(\mathbf{vc}, \mathbf{K})$, ("tally data", ee, vcs), s)
▷ See crypto primitives specification

Output:

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

Verification 2.08 VerifySignatureElectionEventContext

Context:

The trust store containing the system's certificates

Input:

The message ElectionEventContext from table 3
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("sdm_config", ElectionEventContext, ("election event context", ee), s)
▷ See crypto primitives specification
▷ For ElectionEventContext see the System specification, sections 3.3 and 3.4

Output:

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

3.3 Setup - Consistency

Verification 3.01 VerifyEncryptionGroupConsistency

Input:

The encryption group parameters included in the following files from table 2:

- Election Event Context
- Encryption Parameters
- Online Control Components Public Keys ▷ 1 per component
- Setup Component Verification Data ▷ 1 per verification card set and chunk
- Control Component Code Shares ▷ 1 per verification card set and chunk
- Setup Component Tally Data ▷ 1 per verification card set

Output:

$\top$ if all encryption group parameters are identical, $\bot$ otherwise.

Verification 3.02 VerifySetupFileNamesConsistency

Input:

The files from the audit archive in table 2

Operation:

Check that the file names match the paths in the directory of the audit archive

Output:

$\top$ if the file names are consistent, $\bot$ otherwise.

Verification 3.03 VerifyCCRChoiceReturnCodesPublicKeyConsistency

Input:

The CCR Choice Return Codes encryption public keys ($\mathbf{pk}_{\mathrm{CCR}_j}$) included in the following files from table 2:

- 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, $\bot$ otherwise.

Verification 3.04 VerifyCcmElectionPublicKeyConsistency

Input:

The CCM election public keys ($\mathbf{pk}_{\mathrm{CCR}_j}$) included in the following files from table 2:
- 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, $\bot$ otherwise.

Verification 3.05 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 3.06 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 3.07 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 3.08 VerifyPrimesMappingTableConsistency

Input:

- Election Event Context ▷ See table 2

Operation:

1: Verify that the same encoded voting option $\tilde{\mathrm{p}}_i$ maps to the same actual voting option v_i in all verification card sets.

Output:

$\top$ if the primes mapping tables pTable in all verification card sets are consistent, $\bot$ otherwise.

Verification 3.09 VerifyElectionEventIdConsistency

Input:

The election event ID included in the files from table 2.

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 3.10 VerifyVerificationCardSetIdsConsistency

Input:

The verification card set IDs included in the files from table 2.

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 audit archive match the verification card set ID within the files.

Output:

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

Verification 3.11 VerifyFileNameVerificationCardSetIdsConsistency

Input:

The election event context from table 2.
The paths of the audit archive.

Operation:

1: Verify that path names of the audit archive match the list of verification card IDs in the election event context.

Output:

$\top$ if the verification card set IDs between the audit archive and the election event context are consistent, $\bot$ otherwise.

Verification 3.12 VerifyVerificationCardIdsConsistency

Input:

The verification card IDs included in the files from table 2.

Operation:

1: Verify that the verification card IDs $\texttt{vc}_{\texttt{id}}$ match in content and order across all files and that there are no duplicate verification card IDs.

Output:

$\top$ if all verification Card IDs are consistent and in the right order, $\bot$ otherwise.

Verification 3.13 VerifyTotalVotersConsistency

Input:

The following files from table 2:
- Election Event Configuration
- Election Event Context

Operation:

1: Verify that the number of voters in the election event configuration matches the sum of the number of voters in all verification card sets in the election event context.

Output:

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

Verification 3.14 VerifyNodeIdsConsistency

Input:

The node IDs included in the following files from table 2:
- Online Control Component Public Keys ▷ 1 per component
- Online Control Component Code Shares ▷ 1 per component and chunk

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:

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

Verification 3.15 VerifyChunkConsistency

Input:

The chunk IDs included in the following files from table 2:
- Setup Component Verification Data ▷ 1 per chunk
- Online Control Component Code Shares ▷ 1 per component and chunk

Operation:

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

Output:

$\top$ if the chunkIDs are consistent, $\bot$ otherwise.

3.4 Setup - Integrity

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

3.5 Setup - Evidence

The pseudo-code 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.

Verification 5.01 VerifyEncryptionParameters

Input:

Provided group modulus $\hat{p} \in \mathbb{P}$ ▷ See table 2 – Encryption Parameters
Provided group cardinality $\hat{q} \in \mathbb{P}$ s.t. $\hat{p} = 2\hat{q} + 1$ ▷ See table 2 – Encryption Parameters
Provided group generator $\hat{g} \in \mathbb{G}_q$ ▷ See table 2 – Encryption Parameters
$seed \in \mathbb{A}_{UCS}{}^*$ ▷ See table 2 – Encryption Parameters

Require:

Verify that $|\hat{p}|$ corresponds to the *extended* 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 φ voting options does not exceed p to prevent modulo overflow.

Verification 5.02 VerifySmallPrimeGroupMembers

Context:

Group modulus $p \in \mathbb{P}$ ▷ See table 2 – Encryption Parameters
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ ▷ See table 2 – Encryption Parameters
Group generator $g \in \mathbb{G}_q$ ▷ See table 2 – Encryption Parameters
Maximum supported number of voting options $\omega \in \mathbb{N}^*$ ▷ See table 2 – Election Event Context

Input:

The small prime group members in ascending order $\mathbf{p} = (p_0, \ldots, p_{\omega-1}) \in (\mathbb{G}_q \cap \mathbb{P}) \setminus \{2, 3\}, (p_i < p_{i+1})\ \forall\ i \in [0, \omega - 1)$

Operation:

1: $\mathbf{p}' \leftarrow$ GetSmallPrimeGroupMembers(p, q, g, ω) ▷ 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 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_{total} ($n_{total} \geq n$).

Verification 5.03 VerifyVotingOptions

Context:

Group modulus $p \in \mathbb{P}$ ▷ See table 2 – Encryption Parameters
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ ▷ See table 2 – Encryption Parameters
Group generator $g \in \mathbb{G}_q$ ▷ See table 2 – Encryption Parameters
Maximum number of voting options $\omega \in \mathbb{N}^*$
Maximum number of selections $\varphi \in \mathbb{N}^*$
Verification Card Set ID $\texttt{vcs} \in (\mathbb{A}_{Base16})^{\texttt{L}_{\texttt{ID}}}$

Input:

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

Require:

$\varphi \leq \omega$
$0 < n_{total} \leq \omega$

Operation:

1: $\mathbf{p}' \leftarrow (\mathrm{p}_0, \ldots, \mathrm{p}_{n_{total}-1})$
2: **if** $\mathbf{p}' = \tilde{\mathbf{p}}$ **then**
3: verifA $\leftarrow \top$
4: **else**
5: verifA $\leftarrow \bot$
6: **end if**
7: verifB $\leftarrow \prod_{i=(\omega-\varphi)}^{\omega-1} \mathrm{p}_i < p$ ▷ The product of the φ 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, $\bot$ 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.

Verification 5.04 VerifyKeyGenerationSchnorrProofs

Context:

Group modulus $p \in \mathbb{P}$ ▷ See table 2 – Encryption Parameters
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$ ▷ See table 2 – Encryption Parameters
Group generator $g \in \mathbb{G}_q$ ▷ See table 2 – Encryption Parameters
Maximum number of selections $\varphi \in \mathbb{N}^*$
Maximum supported number of write-in options + 1: $\mu \in \mathbb{N}^*$
Number of write-ins for this election event + 1: $\delta \in \mathbb{N}^*$ ▷ See table 2 – Election Event Context
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$ ▷ See table 2 – Election Event Context

Input: ▷ All inputs are taken from table 2 – Setup Component Public Keys

CCR_j Choice Return Codes encryption keys $\left(\mathbf{pk}_{\text{CCR}_1}, \mathbf{pk}_{\text{CCR}_2}, \mathbf{pk}_{\text{CCR}_3}, \mathbf{pk}_{\text{CCR}_4}\right) \in \left(\mathbb{G}_q^{\varphi}\right)^4$
CCR_j Schnorr proofs of knowledge $\left(\boldsymbol{\pi}_{\text{pkCCR},1}, \boldsymbol{\pi}_{\text{pkCCR},2}, \boldsymbol{\pi}_{\text{pkCCR},3}, \boldsymbol{\pi}_{\text{pkCCR},4}\right) \in \left((\mathbb{Z}_q \times \mathbb{Z}_q)^{\varphi}\right)^4$
CCM_j election public keys $\left(\text{EL}_{\text{pk},1}, \text{EL}_{\text{pk},2}, \text{EL}_{\text{pk},3}, \text{EL}_{\text{pk},4}\right) \in \left(\mathbb{G}_q^{\mu}\right)^4$
CCM_j Schnorr proofs of knowledge $\left(\boldsymbol{\pi}_{\text{ELpk},1}, \boldsymbol{\pi}_{\text{ELpk},2}, \boldsymbol{\pi}_{\text{ELpk},3}, \boldsymbol{\pi}_{\text{ELpk},4}\right) \in \left((\mathbb{Z}_q \times \mathbb{Z}_q)^{\mu}\right)^4$
Electoral board public key $\text{EB}_{\text{pk}} \in \mathbb{G}_q^{\delta}$
Electoral board Schnorr proofs of knowledge $\boldsymbol{\pi}_{\text{EB}} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\delta}$

Require:

$\delta \leq \mu \leq \varphi$

Operation:

1: **for** $j \in [1, 4]$ **do**
2: $\quad \mathbf{i}_{\text{aux,CCR},j} \leftarrow (\mathtt{ee}, \text{“}\mathtt{GenKeysCCR}\text{”}, \mathsf{IntegerToString}(j))$
3: $\quad$ **for** $i \in [0, \varphi)$ **do**
4: $\quad\quad \mathsf{VerifSchnorrCCR}_{j,i} \leftarrow \mathsf{VerifySchnorr}(\pi_{\text{pkCCR},j,i}, \text{pk}_{\text{CCR}_j,i}, \mathbf{i}_{\text{aux,CCR},j})$
5: $\quad$ **end for**
6: **end for**
7: **for** $j \in [1, 4]$ **do**
8: $\quad \mathbf{i}_{\text{aux,CCM},j} \leftarrow (\mathtt{ee}, \text{“}\mathtt{SetupTallyCCM}\text{”}, \mathsf{IntegerToString}(j))$
9: $\quad$ **for** $i \in [0, \mu)$ **do**
10: $\quad\quad \mathsf{VerifSchnorrCCM}_{j,i} \leftarrow \mathsf{VerifySchnorr}(\pi_{\text{ELpk},j,i}, \text{EL}_{\text{pk},j,i}, \mathbf{i}_{\text{aux,CCM},j})$
11: $\quad$ **end for**
12: **end for**
13: $\mathbf{i}_{\text{aux,EB}} \leftarrow (\mathtt{ee}, \text{“}\mathtt{SetupTallyEB}\text{”})$
14: **for** $i \in [0, \delta)$ **do**
15: $\quad \mathsf{VerifSchnorrEB}_i \leftarrow \mathsf{VerifySchnorr}(\pi_{\text{EB},i}, \text{EB}_{\text{pk},i}, \mathbf{i}_{\text{aux,EB}})$
16: **end for**
17: **if** $\left(\mathsf{VerifSchnorrCCR}_{j,i}\ \forall\ j, i\right) \wedge \left(\mathsf{VerifSchnorrCCM}_{j',i'}\ \forall\ j', i'\right) \wedge \left(\mathsf{VerifSchnorrEB}_{i''}\ \forall\ i''\right)$ **then**
18: $\quad$ **return** $\top$
19: **else**
20: $\quad$ **return** $\bot$
21: **end if**

Output:

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

The subsequent algorithms check that the control components correctly exponentiated the encrypted partial Choice Return Codes $\mathbf{pCC}_{\mathtt{id}}$ and the encrypted Confirmation Key $\mathtt{CK}_{\mathtt{id}}$. The verifier checks that *all* exponentiation proofs of *all* verification card sets validate. The verifier expects that every verification card set contains at least one verification card.

Verification 5.21 VerifyEncryptedPCCExponentiationProofs

Input:

Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$ ▷ See table 2 – Election Event Context
Vector of verification card set IDs $\mathbf{vcs} = (\mathtt{vcs}_0, \ldots, \mathtt{vcs}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}})^{\mathtt{N_{bb}}}$ ▷ See table 2 – Election Event Context

Operation:

1: **for** $j \in [1, 4]$ **do**
2: **for** $i \in [0, \mathtt{N_{bb}})$ **do**
3: vcsEncryptedPCCVerif$_{j,i} \leftarrow$ VerifyEncryptedPCCExponentiationProofsVerificationCardSet (Context and Input for verification card set $\mathtt{vcs}_\mathtt{i}$ and control component j) ▷ See Algorithm 3.1
4: **end for**
5: **end for**
6: **if** vcsEncryptedPCCVerif$_{j,i}$ $\forall\, j, i$ **then**
7: **return** $\top$
8: **else**
9: **return** $\bot$
10: **end if**

Output:

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

Verification 5.22 VerifyEncryptedCKExponentiationProofs

Input:

Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$ ▷ See table 2 – Election Event Context
Vector of verification card set IDs $\mathbf{vcs} = (\mathtt{vcs}_0, \ldots, \mathtt{vcs}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}})^{\mathtt{N_{bb}}}$ ▷ See table 2 – Election Event Context

Operation:

1: **for** $j \in [1, 4]$ **do**
2: **for** $i \in [0, \mathtt{N_{bb}})$ **do**
3: $\mathsf{vcsEncryptedCKVerif}_{j,i} \leftarrow$ VerifyEncryptedCKExponentiationProofsVerificationCardSet (Context and Input for verification card set $\mathtt{vcs}_\mathtt{i}$ and control component j) ▷ See Algorithm 3.2
4: **end for**
5: **end for**
6: **if** $\mathsf{vcsEncryptedCKVerif}_{j,i}\ \forall\ j, i$ **then**
7: **return** $\top$
8: **else**
9: **return** $\bot$
10: **end if**

Output:

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

3.6 Supporting Algorithms

The verifications in VerifyConfigPhase rely on the following algorithms.

Algorithm 3.1 VerifyEncryptedPCCExponentiationProofsVerificationCardSet

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
The CCR's index $j \in [1, 4]$
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Verification Card Set ID $\mathtt{vcs} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Number or voters $\mathtt{N_E} \in \mathbb{N}^*$
Number of voting options $n \in [0, \omega)$

Input:

Vector of verification card IDs $\mathbf{vc} = (\mathtt{vc}_0, \ldots, \mathtt{vc}_{\mathtt{N_E}-1}) \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}} \times \mathtt{N_E}}$ ▷ See table 2 – Setup Component Verification Data
Encrypted, hashed partial Choice Return Codes $\mathbf{c}_{\mathtt{pCC}} \in (\mathbb{G}_q \times {\mathbb{G}_q}^n)^{\mathtt{N_E}}$ ▷ See table 2 – Setup Component Verification Data
Voter Choice Return Code Generation public keys $\mathbf{K}_j \in {\mathbb{G}_q}^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares
Exponentiated, encrypted, hashed partial Choice Return Codes $\mathbf{c}_{\mathtt{expPCC},j} \in (\mathbb{G}_q \times {\mathbb{G}_q}^n)^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares
Proofs of correct exponentiation $\boldsymbol{\pi}_{\mathtt{expPCC},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares

---

Operation:

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

Output:

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

Algorithm 3.2 VerifyEncryptedCKExponentiationProofsVerificationCardSet

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
The CCR's index $j \in [1, 4]$
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Verification Card Set ID $\mathtt{vcs} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Number or voters $\mathtt{N_E} \in \mathbb{N}^*$

Input:

Vector of verification card IDs $\mathbf{vc} = (\mathtt{vc}_0, \ldots, \mathtt{vc}_{\mathtt{N_E}-1}) \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}} \times \mathtt{N_E}}$ ▷ See table 2 – Setup Component Verification Data
Encrypted, hashed Confirmation Key $\mathbf{c}_{\mathtt{ck}} \in (\mathbb{G}_q \times \mathbb{G}_q)^{\mathtt{N_E}}$ ▷ See table 2 – Setup Component Verification Data
Voter Vote Cast Return Code Generation public keys $\mathbf{Kc}_j \in {\mathbb{G}_q}^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares
Exponentiated, encrypted, hashed Confirmation Key $\mathbf{c}_{\mathtt{expCK},j} \in (\mathbb{G}_q \times \mathbb{G}_q)^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares
Proofs of correct exponentiation $\boldsymbol{\pi}_{\mathtt{expCK},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q)^{\mathtt{N_E}}$ ▷ See table 2 – Control Component Code Shares

---

Operation:

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

Output:

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 data obtained in the VerifyConfigPhase.

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

In addition to the elements needed for the previous phase (see section 3.1), the elements in table 4 should be present.

Description	Path
Control Component Ballot Box	`tally/ballot_boxes/${bb}/controlComponentBallotBoxPayload_${j}.json`
Online Control Component Shuffle	`tally/ballot_boxes/${bb}/controlComponentShufflePayload_${j}.json`
Tally Control Component Shuffle	`tally/ballot_boxes/${bb}/tallyComponentShufflePayload.json`
Tally Control Component Votes	`tally/ballot_boxes/${bb}/tallyComponentVotesPayload.json`
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. 4: The required elements for the final verification

4.2 Final - Authenticity

Table 5 lists the elements that must be signed and the required signers.

Message Name	Signer	Message Content	Context Data
ControlComponentBallotBox	Online CC$_j$	$(\{\mathtt{vc}_{\mathtt{j},\mathtt{i}}, \mathtt{E1}_{j,i}, \widetilde{\mathtt{E1}}_{j,i}, \mathtt{E2}_{j,i}, \pi_{\mathtt{Exp},j,i}, \pi_{\mathtt{EqEnc},j,i}\}_{i=0}^{\mathtt{N_E}-1})$	(“ballotbox”, j, ee, bb)
ControlComponentShuffle	Online CC$_j$	$(\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j})$	(“shuffle”, j, ee, bb)
TallyComponentShuffle	Tally CC	$(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$	(“shuffle”, “offline”, ee, bb)
TallyComponentVotes	Tally CC	$(L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$	(“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 0110”)

Tab. 5: Overview of the authenticity checks for the final verification

See section 3.2 for the signature of XML documents.

Verification 7.01 VerifySignatureControlComponentBallotBox

Context:

The trust store containing the system's certificates

Input:

The message ControlComponentBallotBox from table 5

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

Operation:

1: VerifySignature("control_component_j",
$(\{\mathtt{vc}_{\mathtt{j,i}}, \mathtt{E1}_{j,i}, \widetilde{\mathtt{E1}}_{j,i}, \mathtt{E2}_{j,i}, \pi_{\mathtt{Exp},j,i}, \pi_{\mathtt{EqEnc},j,i}\}_{i=0}^{\mathtt{N_E}-1})$,
("ballotbox", j, ee, bb), s)

▷ See crypto primitives specification

Output:

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

Verification 7.02 VerifySignatureControlComponentShuffle

Context:

The trust store containing the system's certificates

Input:

The message ControlComponentShuffle from table 5

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

Operation:

1: VerifySignature("control_component_j",
$(\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}, \mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j})$,
("shuffle", j, ee, bb), s)

▷ See crypto primitives specification

Output:

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

Verification 7.03 VerifySignatureTallyComponentShuffle

Context:

The trust store containing the system's certificates

Input:

The message TallyComponentShuffle from table 5

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

Operation:

1: VerifySignature("sdm_tally", $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$, ("shuffle", "offline", ee, bb), s)
▷ See crypto primitives specification

Output:

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

Verification 7.04 VerifySignatureTallyComponentVotes

Context:

The trust store containing the system's certificates

Input:

The message TallyComponentVotes from table 5

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

Operation:

1: VerifySignature("sdm_tally", $(L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$, ("decoded votes", ee, bb), s)
▷ See crypto primitives specification

Output:

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

Verification 7.05 VerifySignatureTallyComponentDecrypt

Context:

The trust store containing the system's certificates

Input:

The message TallyComponentDecrypt from table 5

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

Operation:

1: VerifySignature("sdm_tally", evoting decrypt XML, ("evoting decrypt"), s)
▷ See crypto primitives specification

Output:

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

Verification 7.06 VerifySignatureTallyComponentEch0222

Context:

The trust store containing the system's certificates

Input:

The message TallyComponentEch0222 from table 5
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("sdm_tally", eCH 0222 XML, ("eCH 0222"), s)
▷ See crypto primitives specification

Output:

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

Verification 7.07 VerifySignatureTallyComponentEch0110

Context:

The trust store containing the system's certificates

Input:

The message TallyComponentEch0110 from table 5
The signature $s \in \mathcal{B}^*$

Operation:

1: VerifySignature("sdm_tally", eCH 0110 XML, ("eCH 0110"), s)
▷ See crypto primitives specification

Output:

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

4.3 Final - Consistency

Verification 8.01 VerifyConfirmedEncryptedVotesConsistency

Input:

The following files from the audit archive in table 4:
- Control Component Ballot Box ▷ 1 per component

Operation:

1: Verify that the confirmed encrypted votes are identical across all control components—order notwithstanding.

Output:

$\top$ if the confirmed encrypted votes are consistent, $\bot$ otherwise.

Verification 8.02 VerifyCiphertextsConsistency

Input:

The election event context from table 2.

The following files from the audit archive in table 4:

- 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.

Output:

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

Verification 8.03 VerifyPlaintextsConsistency

Input:

The election event context from table 2.

The tally control component shuffle table 4.

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:

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

Verification 8.04 VerifyVerificationCardIdsConsistency

Input:

The verification card IDs from the setup component tally data from table 2.

The election event context from table 2 to map the verification card set ID to the ballot box ID.

The verification card IDs from the control component ballot box from table 4. ▷ 1 per component

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.

Output:

$\top$ if all ballot boxes contain the expected verification card IDs, $\bot$ otherwise.

Verification 8.05 VerifyBallotBoxIdsConsistency

Input:

The ballot box IDs included in the files from table 4.

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 audit archive match the ballot box ID within the files.

Output:

$\top$ if the ballot box IDs are consistent, $\bot$ otherwise.

Verification 8.06 VerifyFileNameBallotBoxIdsConsistency

Input:

The election event context from table 4.
The paths of the audit archive.

Operation:

1: Verify that path names of the audit archive match the list of ballot box IDs in the election event context.

Output:

$\top$ if the ballot box IDs between the audit archive and the election event context are consistent, $\bot$ otherwise.

Verification 8.07 VerifyNumberConfirmedEncryptedVotesConsistency

Input:

The following files from the audit archive in table 4:

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

Operation:

1: Verify that the number of confirmed votes is identical in all files

Output:

$\top$ if the number of confirmed votes is consistent, $\bot$ otherwise.

Verification 8.08 VerifyElectionEventIdConsistency

Input:

The election event ID from the election event context - see table 2.
The election event ID included in the files from table 4.

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 8.09 VerifyNodeIdsConsistency

Input:

The node IDs included in the following files from table 4:
- Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ▷ 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:

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

Verification 8.10 VerifyFileNameNodeIdsConsistency

Input:

The following files from the audit archive in table 4:
- Control Component Ballot Box ▷ 1 per component
- Online Control Component Shuffle ▷ 1 per component

Operation:

Check that the file names match the paths in the directory of the audit archive

Output:

$\top$ if the file names are consistent, $\bot$ otherwise.

Verification 8.11 VerifyEncryptionGroupConsistency

Input:

The election event context - see table 2.
The files from table 4.

Output:

$\top$ if all encryption group parameters are identical, $\bot$ otherwise.

4.4 Final - Integrity

All integrity checks are performed as part of the domain definitions of the pseudo-code 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 $\mathsf{GetMixnetInitialCiphertexts}_1$ 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.

VerifyTally succeeds *only* if the validation of *all* ballot boxes succeed.

Verification 10.01 VerifyOnlineControlComponents

Input:

Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Vector of ballot box IDs $\mathbf{bb} = (\mathtt{bb}_0, \ldots, \mathtt{bb}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}})^{\mathtt{N_{bb}}}$
Number of selectable voting options $\psi \in [1, 120]$
First Control Component Ballot Boxes ▷ See table 4
Online Control Component Shuffles ▷ See table 4
Setup Component Tally Data ▷ See table 2
Election Event Context ▷ See table 2
Setup Component Public Keys ▷ See table 2

Operation:

1: **for** $i \in [0, \mathtt{N_{bb}})$ **do**
2: Extract the key-value map of the verification card public keys **KMap** from the Setup Component Tally Data
3: Prepare the *Context* including the election event context and setup component public keys
4: Prepare the *Input* containing **KMap**, the first control component's ballot box, and the online control component shuffles
5: $\mathsf{bbOnlineCCVerif}_i \leftarrow$ VerifyOnlineControlComponentsBallotBox(Context and Input for ballot box $\mathtt{bb_i}$) ▷ See Algorithm 4.1
6: **end for**
7: **if** $\mathsf{bbOnlineCCVerif}_i \; \forall \; i$ **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 4.1 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 $\mathtt{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$

Setup Component Public Keys ▷ See table 2

Election public key $\mathtt{EL_{pk}} = (\mathtt{EL_{pk},0}, \ldots, \mathtt{EL_{pk},\delta-1}) \in \mathbb{G}_q^{\delta}$
CCM election public keys $(\mathtt{EL_{pk},1}, \mathtt{EL_{pk},2}, \mathtt{EL_{pk},3}, \mathtt{EL_{pk},4}) \in \mathbb{G}_q^{\mu} \times \mathbb{G}_q^{\mu} \times \mathbb{G}_q^{\mu} \times \mathbb{G}_q^{\mu}$
Electoral board public key $\mathtt{EB_{pk}} \in \mathbb{G}_q^{\delta}$
Choice Return Codes encryption public key $\mathbf{pk}_{\mathrm{CCR}} \in \mathbb{G}_q^{\varphi}$

Election Event Context ▷ See table 2

Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Number of selectable voting options $\psi \in [1, 120]$ ▷ see system specification
Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$ ▷ see system specification
Number of eligible voters $\mathtt{N_E} \in \mathbb{N}^*$ ▷ see system specification
Primes Mapping Table $\mathsf{pTable} = (\tilde{\mathbf{v}}, \tilde{\mathbf{p}}) \in \left(\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g)\right)^n$ ▷ see system specification

Input:

Key-value map of the verification card public keys $\mathbf{KMap} = \left((\mathtt{vc}_0, \mathtt{K}_0), \ldots, (\mathtt{vc}_{\mathtt{N_E}-1}, \mathtt{K}_{\mathtt{N_E}-1})\right) \in \left((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}} \times \mathbb{G}_q\right)^{\mathtt{N_E}}$

First Control Component Ballot Box ▷ See table 4

- Control component's list of confirmed verification card IDs $\mathbf{vc_1} = (\mathtt{vc}_{1,0}, \ldots, \mathtt{vc}_{1,\mathtt{N_C}-1}) \in \left((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}\right)^{\mathtt{N_C}}$
- Control component's list of encrypted, confirmed votes $\mathbf{E1}_1 = (\mathtt{E1}_{1,0}, \ldots, \mathtt{E1}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q \times {\mathbb{G}_q}^{\hat{\delta}}\right)^{\mathtt{N_C}}$
- Control component's list of exponentiated, encrypted, confirmed votes $\widetilde{\mathbf{E1}}_1 = (\widetilde{\mathtt{E1}}_{1,0}, \ldots, \widetilde{\mathtt{E1}}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q \times \mathbb{G}_q\right)^{\mathtt{N_C}}$
- Control component's list of encrypted, partial Choice Return Codes $\mathbf{E2}_1 = (\mathtt{E2}_{1,0}, \ldots, \mathtt{E2}_{1,\mathtt{N_C}-1}) \in \left(\mathbb{G}_q \times {\mathbb{G}_q}^{\psi}\right)^{\mathtt{N_C}}$
- Control component's list of exponentiation proofs $\boldsymbol{\pi}_{\mathtt{Exp},1} = (\pi_{\mathtt{Exp},1,0}, \ldots, \pi_{\mathtt{Exp},1,\mathtt{N_C}-1}) \in \left(\mathbb{Z}_q \times \mathbb{Z}_q\right)^{\mathtt{N_C}}$
- Control component's list of plaintext equality proofs $\boldsymbol{\pi}_{\mathtt{EqEnc},1} = (\pi_{\mathtt{EqEnc},1,0}, \ldots, \pi_{\mathtt{EqEnc},1,\mathtt{N_C}-1}) \in \left(\mathbb{Z}_q \times {\mathbb{Z}_q}^2\right)^{\mathtt{N_C}}$

Control Component Shuffles ▷ See table 4

- Preceding shuffled votes $\{\mathbf{c}_{\mathsf{mix},j}\}_{j=1}^{4} \in \left((\mathbb{H}_l)^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$
- Preceding shuffle proofs $\{\pi_{\mathsf{mix},j}\}_{j=1}^{4}$ ▷ See the domain of the Shuffle proof
- Preceding partially decrypted votes $\{\mathbf{c}_{\mathsf{dec},j}\}_{j=1}^{4} \in \left((\mathbb{H}_l)^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$
- Preceding decryption proofs $\{\boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^{4} \in \left((\mathbb{Z}_q \times {\mathbb{Z}_q}^l)^{\hat{\mathtt{N}}_\mathtt{C}}\right)^4$

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

Operation:

1: Extract the key-value map of verification card IDs to encrypted, confirmed votes from the first Control Component Ballot Box $\mathbf{vcMap}_1 = \left((\mathtt{vc}_{1,0}, \mathtt{E1}_{1,0}) \ldots, (\mathtt{vc}_{1,\mathtt{N_C}-1}, \mathtt{E1}_{1,\mathtt{N_C}-1})\right) \in \left((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}} \times \mathbb{H}_l\right)^{\mathtt{N_C}}$
2: **if** $\mathtt{N_C} \geq 1$ **then** ▷ Verifying the voting client proofs requires at least one confirmed vote
3: vcProofsVerif ← VerifyVotingClientProofs($\mathbf{vc_1}, \mathbf{E1}_1, \widetilde{\mathbf{E1}}_1, \mathbf{E2}_1, \boldsymbol{\pi}_{\mathtt{Exp},1}, \boldsymbol{\pi}_{\mathtt{EqEnc},1}, \mathbf{KMap}, \mathtt{EL_{pk}}, \mathbf{pk}_{\mathrm{CCR}}$) ▷ see system specification
4: **else**
5: vcProofsVerif ← $\top$
6: **end if**
7: $\mathbf{c}_{\mathsf{init},1}$ ← GetMixnetInitialCiphertexts$_1(\hat{\delta}, \mathbf{vcMap}_1, \mathtt{EL_{pk}})$ ▷ see system specification
8: shuffleProofsVerif ← VerifyMixDecOffline$\Big(\mathbf{c}_{\mathsf{init},1}, \{\mathbf{c}_{\mathsf{mix},j}\}_{j=1}^{4}, \{\pi_{\mathsf{mix},j}\}_{j=1}^{4}, \{\mathbf{c}_{\mathsf{dec},j}\}_{j=1}^{4}, \{\boldsymbol{\pi}_{\mathsf{dec},j}\}_{j=1}^{4}, \mathtt{EL_{pk}}, (\mathtt{EL_{pk},1}, \mathtt{EL_{pk},2}, \mathtt{EL_{pk},3}, \mathtt{EL_{pk},4}), \mathtt{EB_{pk}}\Big)$ ▷ see system specification
9: **if** vcProofsVerif ∧ shuffleProofsVerif **then**
10: **return** $\top$
11: **else**
12: **return** $\bot$
13: **end if**

Output:

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

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

Verification 10.02 VerifyTallyControlComponent

Input:

Election event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Vector of ballot box IDs $\mathbf{bb} = (\mathtt{bb}_0, \ldots, \mathtt{bb}_{\mathtt{N_{bb}}-1}) \in ((\mathbb{A}_{Base16})^{\mathtt{L_{ID}}})^{\mathtt{N_{bb}}}$
Last Online Control Component Shuffles ▷ See table 4
Tally Control Component Shuffles ▷ See table 4
Tally Control Component Votes ▷ See table 4
Election Event Context ▷ See table 2
Setup Component Public Keys ▷ See table 2
Election Event Configuration ▷ See table 2
Tally Control Component Decryptions ▷ See table 4
Tally Control Component Results ▷ See table 4

Operation:

1: **for** $i \in [0, \mathtt{N_{bb}})$ **do**
2: Prepare the *Context* containing the parameters from the election event context and the setup component public keys including the primes mapping table pTable
3: Extract the partially decrypted votes $\mathbf{c}_{\mathsf{dec},4}$ from the last control component's shuffle
4: Extract $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$ from the Tally control component shuffle
5: Extract the list of selected encoded voting options L_{votes}, selected actual voting options $L_{\mathsf{decodedVotes}}$, and selected decoded write-in votes L_{writeIns} from the Tally control component votes
6: $\mathit{Input} \leftarrow (L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}}, \mathbf{c}_{\mathsf{dec},4}, \mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5})$
7: $\mathsf{tallyVerif}_i \leftarrow$ VerifyTallyControlComponentBallotBox(Context and Input for ballot box $\mathtt{bb_i}$) ▷ See Algorithm 4.2
8: **end for**
9: tallyFilesVerif ← VerifyTallyFiles(Context and input as specified) ▷ See Algorithm 4.4
10: **if** $\mathsf{tallyVerif}_i \; \forall \; i \wedge \mathsf{tallyFilesVerif}$ **then**
11: **return** $\top$
12: **else**
13: **return** $\bot$
14: **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 $\mathtt{bb} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Electoral board public key $\mathtt{EB_{pk}} \in \mathbb{G}_q^{\delta}$
Primes Mapping Table $\mathsf{pTable} = (\tilde{\mathbf{v}}, \tilde{\mathbf{p}}) \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g))^n$ ▷ see system specification
Write-in voting options $\tilde{\mathbf{p}}_{\mathrm{w}} = (\tilde{\mathrm{p}}_{\mathrm{w},0}, \ldots, \tilde{\mathrm{p}}_{\mathrm{w},\hat{\delta}-2}) \in \left((\mathbb{G}_q \cap \mathbb{P}) \setminus g\right)^{\hat{\delta}-1}$
Number of selectable voting options $\psi \in [1, 120]$ ▷ see system specification
Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$ ▷ see system specification

Input:

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

Require: $l = \hat{\delta}$
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 $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N}_\mathtt{C} + 2$ if $\mathtt{N}_\mathtt{C} < 2$
Require: $\hat{\mathbf{p}}_i \subseteq \tilde{\mathbf{p}}, \forall i \in \{0, \ldots, (\mathtt{N}_\mathtt{C} - 1)\}$
Require: $\hat{\mathrm{p}}_{i,k} \neq \hat{\mathrm{p}}_{i,l}, \forall i \in \{0, \ldots, (\mathtt{N}_\mathtt{C} - 1)\}, \forall k, l \in \{0, \ldots, (\psi - 1)\} \wedge k \neq l$ ▷ 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: $\mathtt{EB}_{\mathtt{pk,cut}} \leftarrow (\mathtt{EB}_{\mathtt{pk},0}, \cdots, \mathtt{EB}_{\mathtt{pk},\hat{\delta}-1})$
3: $\mathsf{shuffleVerif} \leftarrow \mathsf{VerifyShuffle}(\mathbf{c}_{\mathsf{dec},4}, \mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathtt{EB}_{\mathtt{pk,cut}})$ ▷ See crypto primitives specification
4: $\mathsf{decryptVerif} \leftarrow \mathsf{VerifyDecryptions}(\mathbf{c}_{\mathsf{mix},5}, \mathtt{EB}_{\mathtt{pk,cut}}, \mathbf{m}, \boldsymbol{\pi}_{\mathsf{dec},5}, \mathbf{i}_{\mathsf{aux}})$ ▷ See crypto primitives specification
5: $\mathsf{processVerif} \leftarrow \mathsf{VerifyProcessPlaintexts}(\mathsf{pTable}, \mathbf{m}, \tilde{\mathbf{p}}_{\mathrm{w}}, \psi, \hat{\delta}, L_{\mathsf{votes}}, L_{\mathsf{decodedVotes}}, L_{\mathsf{writeIns}})$ ▷ See algorithm 4.3
6: **if** $\mathsf{shuffleVerif} \wedge \mathsf{decryptVerif} \wedge \mathsf{processVerif}$ **then**
7: **return** $\top$
8: **else**
9: **return** $\bot$
10: **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 $g \in \mathbb{G}_q$
Election event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$
Ballot box ID $\mathsf{bb} \in (\mathbb{A}_{Base16})^{\mathtt{L_{ID}}}$

Input:

Primes Mapping Table $\mathsf{pTable} = (\tilde{\mathbf{v}}, \tilde{\mathbf{p}}) \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g))^n$ ▷ see system specification
List of plaintext votes $\mathbf{m} = (m_0, \dots m_{\hat{\mathtt{N}}_\mathtt{C}-1}) \in ({\mathbb{G}_q}^l)^{\hat{\mathtt{N}}_\mathtt{C}}$
Write-in voting options $\tilde{\mathbf{p}}_\mathrm{w} = (\tilde{\mathrm{p}}_{\mathrm{w},0}, \dots, \tilde{\mathrm{p}}_{\mathrm{w},\hat{\delta}-2}) \in \left((\mathbb{G}_q \cap \mathbb{P}) \setminus g\right)^{\hat{\delta}-1}$
Number of selectable voting options $\psi \in [1, 120]$ ▷ see system specification
Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$ ▷ see system specification
List of all selected encoded voting options $L_{\mathsf{votes}} = (\hat{\mathbf{p}}_0, \dots, \hat{\mathbf{p}}_{\mathtt{N}_\mathtt{C}-1}) \in \left(((\mathbb{G}_q \cap \mathbb{P}) \setminus g)^\psi\right)^{\mathtt{N}_\mathtt{C}}$ ▷ 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 $L_{\mathsf{decodedVotes}} = (\hat{\mathbf{v}}_0, \dots, \hat{\mathbf{v}}_{\mathtt{N}_\mathtt{C}-1}) \in \left((\mathcal{T}_1^{50})^\psi\right)^{\mathtt{N}_\mathtt{C}}$
List of all selected decoded write-in votes $L_{\mathsf{writeIns}} = (\hat{\mathbf{s}}_0, \dots, \hat{\mathbf{s}}_{\mathtt{N}_\mathtt{C}-1}) \in \left((\mathbb{A}^*_{\mathsf{writein}})^*\right)^{\mathtt{N}_\mathtt{C}}$

Require: $\hat{\delta} = l$
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 $\hat{\mathtt{N}}_\mathtt{C} = \mathtt{N}_\mathtt{C} + 2$ if $\mathtt{N}_\mathtt{C} < 2$

Operation:

1: $\vec{1} \leftarrow \underbrace{(1, \dots, 1)}_{\hat{\delta} \text{ times}}$
2: $k \leftarrow 0$
3: **for** $i \in [0, \hat{\mathtt{N}}_\mathtt{C})$ **do**
4: $\quad m_i = (\phi_{i,0}, \dots, \phi_{i,l-1})$
5: $\quad$ **if** $m_i \neq \vec{1}$ **then**
6: $\quad\quad \hat{\mathbf{p}}'_k \leftarrow \mathsf{Factorize}(\phi_{i,0}, \tilde{\mathbf{p}}, \psi)$ ▷ see system specification
7: $\quad\quad \hat{\mathbf{v}}'_k \leftarrow \mathsf{DecodeVotingOptions}(\hat{\mathbf{p}}'_k, \mathsf{pTable})$ ▷ see system specification
8: $\quad\quad \mathbf{w}'_k \leftarrow (\phi_{i,1}, \dots, \phi_{i,l-1})$ ▷ An empty vector if the election event has no write-ins
9: $\quad\quad \hat{\mathbf{s}}'_k \leftarrow \mathsf{DecodeWriteIns}(\tilde{\mathbf{p}}_\mathrm{w}, \hat{\mathbf{p}}'_k, \mathbf{w}'_k)$ ▷ see system specification
10: $\quad\quad k \leftarrow k + 1$
11: $\quad$ **end if**
12: **end for**
13: **if** $\left((\hat{\mathbf{p}}'_0, \dots, \hat{\mathbf{p}}'_{\mathtt{N}_\mathtt{C}-1}) = L_{\mathsf{votes}}\right) \wedge \left((\hat{\mathbf{v}}'_0, \dots, \hat{\mathbf{v}}'_{\mathtt{N}_\mathtt{C}-1}) = L_{\mathsf{decodedVotes}}\right) \wedge \left((\hat{\mathbf{s}}'_0, \dots, \hat{\mathbf{s}}'_{\mathtt{N}_\mathtt{C}-1}) = L_{\mathsf{writeIns}}\right)$ **then**
14: $\quad$ **return** $\top$
15: **else**
16: $\quad$ **return** $\bot$
17: **end if**

Output:

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

Algorithm 4.4 VerifyTallyFiles

Context:

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

Input:

Configuration file configuration XML

File aggregating the submitted votes by ballot box id evoting decrypt XML

File containing the tallied votes eCH 0110 XML

For each ballot box, the list of all selected decoded voting options $L_{\mathsf{decodedVotes_{bb}}} \in \left((\mathcal{T}_1^{50})^{\psi}\right)^{\mathtt{Nc_{bb}}}$

Operation:

1: evoting decrypt XML′ ← Aggregate the decoded voting options from all ballot boxes
2: eCH 0110 XML′ ← Count how many votes each voting option received, in the format defined under http://www.ech.ch/xmlns/eCH-0110/4/eCH-0110-4-0.xsd.
3: **return** evoting decrypt XML = evoting decrypt XML′ ∧ eCH 0110 XML = eCH 0110 XML′

Output:

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

As noted in the system specification, the details of the rules taken into account for the constitution of both XML files are beyond the scope of this document, but the schemas are attached or referenced 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] S. Post: *Protocol of the Swiss Post Voting System. Computational Proof of Complete Verifiability and Privacy. Version 1.1.0*. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Protocol. Dec. 2022.

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

List of Algorithms

List of Verifications

List of Figures

List of Tables