Berner Fachhochschule (BFH), CH-2501 Biel, Switzerland

Examination of the Swiss Post Internet Voting System

Scope 2: Software

Rolf Haenni, Reto E. Koenig, Philipp Locher, Eric Dubuis

March 28, 2022

On behalf of the Federal Chancellery

Revision History

Revision	Date	Description
0.1	14.08.2021	Document initialization.
0.2	28.11.2021	Draft submitted to Federal Chancellery.
1.0	28.03.2022	Final version submitted to Federal Chancellery. Some clarifications added after receiving feedback from the Federal Chancellery and from Swiss Post. Minor textual improvements.

Contents

Management Summary **4**

1 Introduction **6**
1.1 Relevant Documents 6
1.2 Source Code 8
1.3 Purpose and Scope of Examination 9
1.4 Summary of Findings 10

2 Topics of Examination **13**
2.1 Deviations Between Protocol and System Specification 13
2.2 Deviations Between System Specification and Source Code 18
2.3 Crypto-Primitives 22
2.4 Underspecified Concepts and System Components 24
2.5 Quality of Code 26
2.6 Synchronization 29
2.7 Randomness 30

Management Summary

This report is the main output from our analysis of the current implementation of the Swiss Post e-voting system. Compared to the documents and code released in 2019, we observed numerous improvements in many places. We can also confirm that the development team at Swiss Post makes a great effort in aligning the source code with the cryptographic protocol. There are areas in which this alignment has reached a satisfactory degree, but there are also areas, in which unfortunately this is not yet the case. Overall, this gives the impression that we have been given an unfinished project to look at, which generally points into the right direction, but which has not yet reached the state that one would expect when doing such an assessment. One obvious reason for this unfortunate situation is the underlying cryptographic protocol, which itself still seems to be relatively unstable at this moment. The fact that several software updates have been released during our assessment underlines our impression of looking at work in progress.

In the light of the above remarks, we can neither confirm nor decline that the software will at some point fulfill the legal requirements of the draft OEV. While we are incapable of making a final verdict at this stage of the approval process, we expect that the output of our analysis will serve useful for the system developers at Swiss Post to further improve the quality of their product. Together with the improvements suggested by other experts, this may represent an important step forward towards a system that at some point will fulfill the legal requirements and pass the approval process, even if this process will need more time than expected.

A very positive aspect of our assessment comes from the fact that the software is now owned and developed as an internal project at Swiss Post. Contacting the right person for getting a quick answer to a specific question has always been very efficient and uncomplicated. All members of the Swiss Post development team showed their eagerness to help and learn from our questions and feedback. This demonstrates that Swiss Post has made great progress in implementing their new strategy of being as open and transparent as possible. This creates an entirely new working atmosphere, from which both the people from Swiss Post and the external auditors greatly benefit.

On the more technical side, we did not discover evident programming errors that could directly be exploited by an adversary for conducting an attack on either the integrity or privacy of the votes. However, we found several problematical areas, in which the current implementation starts from questionable assumptions. One such area is the synchronization of ballots in situations where multiple ballots are submitted in parallel by the same voter. Here the implementation is based on the assumption that such situations will never occur in practice. Another example is the delegation of setting up a reliable randomness source to the operating system of the machines that will execute the software. Here it is assumed that the administrator of these machines will always be able to guarantee the reliability of the machine's randomness source. Since failures may

have devastating consequences for the overall security in both examples, we believe that the underlying assumptions are unacceptably strong.

1 Introduction

This report lists the findings of our assessment of the Swiss Post e-voting system implementation. We have conducted this assessment in parallel to analyzing the underlying cryptographic protocol. Both tasks—called *Scope 1* and *Scope 2*—have been assigned to us by the Federal Chancellery in June 2021 for a period of 6 months. While it is evident that the cryptographic protocol and the system implementation are closely interlinked, we tried to separate the two scopes in our work as much as possible. This means that in this report we do not discuss findings that exist in both scopes, if they have already been discussed in the other report. By doing so, the two resulting reports are largely independent, but we still recommend looking at them as two complementary documents about the same topic.

A draft of this document has been given to the Federal Chancellery on November 28, 2021. The feedback that we received from the Federal Chancellery in January 2022 and from Swiss Post in March 2022 allowed us to finalize the document by inserting a few clarifying remarks in various places. The content of this document has been worked out jointly by the listed authors from the Bern University of Sciences and independently of any other group of people. During our mission, we have been in loose contact with the Swiss Post, mainly for obtaining clarifying information on certain topics. All contacted members of the Swiss Post development team showed their eagerness to answer our questions efficiently and to learn from our preliminary feedback.

There were also two general meetings with all involved experts, a *General Overview Meeting* on August 19 and an *Audit Planning Meeting* on September 3. All the documents related to these meetings (slides, meeting notes, etc) were given to us in a timely manner.

1.1 Relevant Documents

To conduct our assessment, we received two relevant documents from the Federal Chancellery, the legal ordinance with its annex and an explanatory report with additional clarifying information:

- [DraftOEV] *Federal Chancellery Ordinance on Electronic Voting*, Federal Chancellery FCh, Draft of April 28, 2021 (with Annex on Technical and Administrative Requirements for Electronic Voting).
- [ExpRep] *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 Consultation*, Federal Chancellery FCh, April 28, 2021.

During the writing of this report, [DraftOEV] went through a public consultation process. According to a press release on December 10, 2021, the Federal Council has decided to

finalize and publish the new ordinance in mid-2022. Given the large amount of responses to the consultation, final documents are likely to contain some changes.

To emphasize its state as a non-final document, we refer to it as "draft OEV" (as opposed to "current OEV", which we will sometimes use to refer to the current ordinance from 2018). For understanding the requirements defined in [DraftOEV] and [ExpRep] as precisely as possible, we have mainly looked at the official document versions in German. However, the terminology and citations used in this document are all taken from the available English translations.

Swiss Post has released many documents describing various aspects of their system and its development process. Some of them were of minor importance for conducting our assessment. The most relevant documents were the following:

- [ProtSpec] *Protocol of the Swiss Post Voting System – Computational Proof of Complete Verifiability and Privacy*, Version 0.9.10, Swiss Post Ltd., June 25, 2021.
- [SysSpec] *Swiss Post Voting System – System Specification*, Version 0.9.6, Swiss Post Ltd., June 25, 2021.
- [CryptoPrim] *Cryptographic Primitives of the Swiss Post Voting System – Pseudo-Code Specification*, Version 0.9.6, Swiss Post Ltd., July 26, 2021.
- [ArchDoc] *SwissPost Voting System – Architecture Document*, Version 0.9.1, Swiss Post Ltd., August 17, 2021.
- [VerifSpec] *Swiss Post Voting System – Verifier Specification*, Version 0.9, Swiss Post Ltd., September 1, 2021.
- [DevProc] *Software Development Process for the Swiss Post Voting System*, Swiss Post Ltd., November 15, 2021.

These documents are all publicly available, either at the Swiss Post *E-Voting Community Programme* web page or in corresponding sub-directories of two interlinked GitLab repositories at

- https://evoting-community.post.ch/en/community-programme,
- https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation,
- https://gitlab.com/swisspost-evoting/verifier/verifier.

The first three documents in the above list have been updated during the assessment period. Due to the given time constraints and strict deadlines and after having invested a significant amount of work into the versions that were given to us in the beginning of our mission, we decided not to consider the updates in our analysis. As the other three documents were only released during the assessment period, we considered them on a best effort basis with the restricted time left at the end of our mission. Generally, we would have expected to receive finalized versions of all documents at the beginning of the mission.

1.2 Source Code

We received Version 0.9.0.0 of the source code on June 29 in a GitLab repository "`UP 2021/Audit Scope 2`" created especially for the purpose of conducting the audits in Scope 2. Three updates 0.9.1.0, 0.9.2.0, and 0.10.0.0 were uploaded during the month of July. Version 0.10.0.0 was for a long time the version that we looked at to conduct our analysis. No further versions have been uploaded to this repository.

In August 2021, Swiss Post started to release the source code in the same public GitLab group in which the documentations have been released in July:

- https://gitlab.com/swisspost-evoting/e-voting/e-voting,
- https://gitlab.com/swisspost-evoting/e-voting/evoting-e2e-dev,
- https://gitlab.com/swisspost-evoting/crypto-primitives,
- https://gitlab.com/swisspost-evoting/verifier/verifier.

During the main period of our mission from July to November 2021, the initial public Version 0.11.3.0 from August 31 has been updated four times into Version 0.11.4.0 on October 7, into Version 0.12.0.0 on November 9, into Version 0.12.0.1 on November 15, and into Version 0.12.0.2 on November 17. Obtaining regular code updates during our mission was increasingly challenging, because it meant to reexamine areas of the code that we already analyzed. After submitting our draft report to the Federal Chancellery on November 28, further updates have been uploaded to the repository. Version 0.13.0.0 and its successor Version 0.13.0.1 from February 23, 2022, contain several major improvements made in response to the evaluation reports submitted to the Federal Chancellery, including this one.

On September 28, 2021, to ensure the consistency of the ongoing examinations with one another, the Federal Chancellery instructed all involved experts to use the code that was available by then. Corresponding version numbers, release dates, and SHA-1 commit fingerprints are listed in the following Table 1. We decided to strictly follow the instructions received from the Federal Chancellery. All the statements made in this report are therefore tied to the versions of the following table.

Code Library	Version	Date	Commit (SHA-1 Fingerprint)
E-voting	0.11.3.0	August 31	`f4b8c2f45970678650115b2e8bf3aeb924ddb05a`
Crypto-primitives	0.11.3.0	August 31	`b97c82394135edac9b53368cf35f0613fb8071ea`
evoting-e2e-dev	0.11.3.0	August 31	`e0a7c8da5b7601c6eeb512efe8c8a644021a2920`
Verifier	0.9.0.0	September 2	`56de7c47cd3daad42bf6bf7a214b99838f1ad864`

Table 1: Library versions as examined in this report.

To whole system code base is a huge collection of files. The core library E-voting, for example, consists of 2'458 Java and 326 JavaScript files with a total of nearly 175'000

actual code lines. Table 2 gives an overview of the code libraries in terms of number of files and number of lines (code, comments, blanks).

Code Library	Language	Files	Lines	Code	Comments	Blanks
E-voting	Java	2458	220947	142236	37064	41647
	JavaScript	326	47541	32569	6892	8080
Crypto-primitives	Java	176	28177	18273	5550	4354
evoting-e2e-dev	Java	83	7632	4631	1576	1425
Verifier	Java	267	23210	15035	4810	3365
	Total:	2458	220947	142236	37064	41647

Table 2: Number of files and code lines in the given libraries.

The files from the core library E-voting are assigned to a total of 55 different Maven projects. We were able to install and build them in our IDE without considerable difficulties. During the building process, dependencies to external libraries were resolved automatically. The possibility of examining the code in our IDE without any broken dependencies was important for conducting our analysis efficiently. Compared to the 2019 code release, this is an important step forward for making the code more accessible to anyone interested in analyzing the code.

1.3 Purpose and Scope of Examination

In a document called "*Audit Concept v1.3*", the Federal Chancellery describes the rules for preparing, conducting and reporting the examination. This document has been given to both the examiners and the examinees. It defines the general purpose of the examination as follows:

> *"In the context of the assessment of the Swiss Post system, the experts shall answer the following questions:*
>
> - *Are the system, its development and operation compliant with the legal requirements [...]?*
> - *Are the measures taken to mitigate risks effective?*
> - *Which improvements could be made for the sake of security, trust and acceptance?"*

The same document also defines the specific goals and examination criteria for Scope 2:

> *"The software of the system including the auditor's technical aid must fulfill the requirements listed in Chapters 2 to 25 of the annex of the draft OEV and adequately support the protocol [...]. The mapping between a requirement in those paragraphs and the place [...] where it is fulfilled shall be provided by*

the examinees before the examination. Functions whose trustworthiness is decisive for the effectiveness of verifiability as per draft OEV, must be examined in detail on the basis of the source code and the cryptographic protocol.

Moreover, a sample of the functional tests documented and executed by the developer are to be executed by the examiners to validate their results. The sample shall be selected by the examiners on the basis of its coverage of security functions and the contribution of these functions to risk mitigation."

To structure the expected work of the examiners and the focus of the assessment in Scope 2, the document describing the audit concept also provides a list of six different topics numbered from a) to f) and links them to the given requirements as defined in [DraftOEV, Annex]. Figure 1 is taken for there

a) Assess the development process	8.12, 17, 24.1.1, 24.1.2, 24.1.3, 24.1.4, 24.1.14, 24.1.15, 24.1.16, 24.1.17, 24.1.18, 24.1.19, 24.1.20, 24.4, 24.5, 25.13.3, 25.13.4
b) Assess the code quality and security	3.2, 14.2, 14.5, 14.6, 15.2, 15.3, 15.4, 24.1.5, 24.1.6, 24.1.10, 24.1.12, 25.8, 25.9, 25.10.5, 25.10.6, 25.10.7, 25.10.8, 25.11, 25.12, 25.13.2, 25.13.5
c) Asses the documentation quality	24.1.5, 24.1.6, 24.1.7, 24.1.8, 24.1.9, 24.1.12, 24.1.13, 25.2, 25.3, 25.4, 25.5, 25.10.2, 25.10.3, 25.10.4
d) Assess the alignment between software development products	24.1.9, 24.1.11, 25.1.3, 25.2.8
e) Assess the implementation of the protocol	2.5, 2.6, 2.12, 3.16, 25.1.2
f) Assess the functionalities	3.12, 2.11, 4, 5.1, 8.10, 9, 10, 11.5, 11.6, 25.7

Figure 1: Examination topics of Scope 2 as defined in "*Audit Concept v1.3*".

Within the given time constraints, it was impossible to conduct our examination on all the topics listed in Figure 1, also because some of the topics lie outside our area of expertise. Therefore, we decided to focus our examination on the Topics b), d), and e), i.e., on assessing the code quality and security, the alignment between protocol, specification, design and source code, and the implementation of the protocol. These are our principal areas of expertise within this examination scope.

1.4 Summary of Findings

The general impression that we received from analyzing the system implementation is the one of looking at an unfinished project. Certain parts of the documentation and code

have been improved considerably compared to earlier versions, in such a way that we would be able to announce a positive verdict if the whole system had reached that same quality level, but this is unfortunately not yet the case. An example that underlines this point is the existence of two cryptographic libraries, an "old one" called cryptolib and a revised "new one" called crypto-primitives. Both libraries are currently in use, sometimes simultaneously by the same algorithm, but it is evident that the old library is a relic from an earlier version and is supposed to disappear when all its functionalities have been shifted to the new one. Another observation emphasizing the project's work-in-progress status is its dependency to outdated external libraries, of which some will even reach the end-of-life status soon, meaning that no further patches or updates will be released. The removal of such libraries has been announced in the project repository's readme.md file, which also lists some other "known issues" and provides a link to an open issue on the repository's issue tracker. The existence of multiple known issues is something that one would not expect to see in an almost finished project that undergoes an examination process.

Another major problem of the current implementation is the incomplete alignment between the protocol and system specifications and between the system specification and the source code. Checking these alignments is where we invested most of our efforts during the examination process. Again, there are revised areas in which the alignment has achieved a satisfactory level, but there are also (seemingly unrevised) areas, where this is not the case. We can therefore confirm that the general development process points into the right direction, but also that the current version is still far from reaching the stage of finalization. Detailed results of examining the alignments are listed in Sections 2.1 and 2.2.

Similar remarks are possible with respect to the overall code quality. From the perspective of a cryptographic audit, the most significant criteria are the code's readability and comprehensibility, which we see as an important precondition for checking the correctness of the implemented cryptographic protocol. We expected that seemingly simple tasks such as locating the code that implements a given aspect of the protocol should not cause too much difficulties. For some of the given pseudo-code algorithms, locating corresponding Java (or JavaScript) methods was indeed relatively simple, for example by searching through the entire project code base for corresponding keywords. In other cases, however, this search was very difficult to conduct, mostly due to confusing class or method names that are not present in the pseudo-code. Here again, it seems that certain parts of the code have been cleaned up carefully, for example by adopting some naming conventions, while other parts of the code have not yet been revised. Additional readability problems are caused by certain dependencies to third-party libraries, which can be useful for implementing repeating coding patterns efficiently and robustly, but which also introduce additional layers of complexity and obscure the overall program flow. For optimized readability, we would recommend to implement the protocol algorithms without the help of such libraries. In Section 2.5, this point will be discussed further and exemplary problems from the code will be given.

From a security perspective, a major concern with the current implementation is the missing synchronization of ballots in situations where multiple ballots are submitted in parallel by the same voter. Here the implementation is based on the assumption that such situations will never occur in practice. While this assumption may hold for ordinary voters using nothing but the provided web interface of the official election portal, it is certainly not true in general, i.e., in the presence of active adversaries who are trying to provoke such situations on purpose. Synchronization problems of that kind are well known in general web applications, and methods to prevent them are not difficult to implement. Without proper synchronization, the involved parties may run into inconsistent states, which then may prevent the voting process from working properly. Or even worse, if an attacker receives responses from sending multiple ballots, this can undermine the verification properties of the underlying cryptographic protocol. We will further discuss this topic in Section 2.6.

We are also concerned about the generation of high-quality randomness, which is a security-critical topic in every cryptographic application. Currently, the implementation takes the simplest route by creating instances of the Java class `SecureRandom` or in JavaScript by calling the function `getRandomValues` from the Web Crypto API. These are the standard cryptographic pseudo-random generators in the respective environments, so there is nothing wrong about using them. However, the problem in the case of Java `SecureRandom` is the fact that its seeding depends on the current configuration of the operating system on which the JRE is running. To presume that a high-entropy seed can always be obtained in this way is a very strong trust assumption, which we do not support. The problem with using the Web Crypto API in JavaScript comes from the fact that in scripting languages, functionalities can easily be overridden, for example by injecting a few code lines as part of another external library. Relatively simple privacy attacks are therefore possible on the voting client. We provide a more detailed discussion of this problem in Section 2.7.

2 Topics of Examination

2.1 Deviations Between Protocol and System Specification

One of the first questions that we addressed in our analysis is the alignment between the two most relevant documents, the protocol specification [ProtSpec] and the system specification [SysSpec]. To answer this question, we scanned the two documents for differences, most importantly with respect to the message flow in the protocol and the computations performed by the involved parties. For this, we took the high-level description of the protocol algorithms from [ProtSpec] as a reference point for checking their alignment with corresponding pseudo-code algorithms from [SysSpec]. This work turned out to be very time-consuming, because it meant to align in each case all the involved variables and corresponding computations. Since there are already quite a few differences in the formal notation, for example by using different mathematical symbols or slightly different variable names for the same objects or quantities, this work was anything but obvious. Another complicating issue came from the fact that the computational steps of an algorithm are often not presented in exactly the same order, which sometimes created a considerable overhead for re-ordering them manually.

We believe that these general problems of describing the algorithms consistently in both documents are mostly unnecessary. For the security of the system, this is nothing critical, but since it complicates the evaluation process, we recommend to improve it further in future versions of the documents. We also suggest to question the necessity of having descriptions of the same algorithms in two different documents, which clearly is highly redundant. With a single description, many of the problems could be avoided entirely without really losing something. This whole section of our report would become obsolete.

2.1.1 General Deviations

We first scanned the system specification and the architecture documents for deviations with respect to the general structure of the cryptographic protocol and the messages exchanged between the parties. In most places, the correspondence is very high, but we also found some important differences.

One issue is related to the OK/NOT-OK message from the auditors to the CCMs in [ProtSpec, Fig. 23]. In our protocol analysis in Scope 1, we already expressed our concern about involving the auditors at the end of the voting phase, as it was unclear to us how this particular message could possibly be implemented. From a private communication with Swiss Post, we learned that in the actual implementation VerifyVotingPhase is executed together with VerifyOnlineTally at the end of the online mixing process in [ProtSpec, Fig. 24], and that the two OK/NOT-OK messages are sent jointly to the electoral board (not the CCMs). Independently of whether this modification in the protocol

flow creates any security problem, it is clearly a deviation between the protocol and its implementation.

Other deviations in the message flow exist in the *Deployment Overview* from [ArchDoc, Fig. 1], where the *verifier* (the software used by the auditors) receives all the information directly from CCM_4, whereas in [ProtSpec, Figs. 19, 23, 24] information is also received from the print office, the voting server, and the other CCMs. Furthermore, according to [ProtSpec, Fig. 24], there is an information flow from the auditors to the CCMs, but no such communication is depicted [ArchDoc, Fig. 1]. The *Deployment Overview* diagram is therefore clearly not aligned with the cryptographic protocol. Similar deviations exist between [ProtSpec, Fig. 24] and [ArchDoc, Fig. 24], which both depict the tally phase. Generally, we think that the dcoument quality of [ArchDoc] should be improved.

A more subtle deviation from the protocol specification results from a seemingly innocent statement in [SysSpec, Sect. Sect. 6.1], which permits the repetition of the mixing process if either VerifyVotingPhase or VerifyOnlineTally detect a failure. Unless explicitly permitted in the protocol specification, repeating certain protocol steps is highly problematical, because it may render existing security proofs invalid. For example, it could happen that an adversary provokes the repetition of these steps on purpose, with the goal of breaking the protocol's security properties.

A very general deviation, that appears in many places, comes from the fact that the possibility of write-in votes has not been taken into account in the protocol specification, whereas the system specification supports the general case of $l - 1$ write-ins with the aid of multi-recipient ElGamal encryptions of size l. This affects the key generation, encryption, decryption and proof generation algorithms, which are therefore not aligned with the protocol specification. In our Scope 1 report, we have already discussed problems related to write-ins and recommended removing them entirely from the protocol. But then they should also be removed from the current implementation.

2.1.2 Deviations Between Algorithms

In the introduction of this section, we already mentioned some of the difficulties that we encountered when comparing the algorithms as described in [ProtSpec] and [SysSpec]. Other unnecessary complications resulted from the fact that algorithms are not numbered and vectors are not indexed consistently across both documents. In the summarizing tables given below, we use the algorithm numbers from [SysSpec] and ignore any indexing differences. As one can see from our results, we discovered quite a lot of deviations—minor and major ones—in almost all algorithms. The requirement from [DraftOEV, Annex 25.2.8], namely that "*The cryptographic protocol, specification, design and source code are aligned*", is therefore certainly not yet fulfilled to a satisfactory degree.

Configuration Phase

4.1 GenKeysCCR$_j$	×	Protocol defines $k'_j \in \mathbb{Z}_q^*$, not $k'_j \in \mathbb{Z}_q$.
4.2 GenVerCardSetKeys	✓	Sufficient alignment reached.
4.3 GenSetupEncryptionKeys	✓	Inconsistent use of symbols n and ω. Otherwise, sufficient alignment reached.
4.4 GenVerDat	×	Inconsistent naming of variables $\mathbf{hpccHash}_{\mathtt{id}}$ and $\mathtt{lpCC}_{\mathtt{id},k}$, $\mathtt{hck}_{\mathtt{id}}$ and $\mathtt{hCK}_{\mathtt{id}}$, BCK and bck, and $\mathtt{vcd}_{\mathtt{id}}$ and $\mathtt{vc}_{\mathtt{id}}$. Compression of public keys undefined in protocol. Protocol defines $\mathsf{L}_{\mathtt{pCC}}$ to be shuffled, not not be ordered alphabetically (shuffling instead of ordering is an important subtlety in the security proof).
4.5 GenEncLongCodeShares$_j$	×	$\mathbf{i}_{\mathtt{aux}}$ is undefined in protocol. Inconsistent naming of variables $\mathtt{kc}_{j,\mathtt{id}}$ and $\mathtt{k}_{j,\mathtt{id}}$.
4.6 CombineEncLongCodeShares	✓	Sufficient alignment reached.
4.7 GenCMTable	×	Usage of key derivation function MGF1 is undefined in protocol. Inconsistent naming of variables $\mathtt{vcd}_{\mathtt{id}}$ and $\mathtt{vc}_{\mathtt{id}}$. Compression of secret keys undefined in protocol. The protocol defines the CMtable to be shuffled, not be ordered alphabetically (shuffling instead of ordering is an important subtlety in the security proof).
4.8 GenCredDat	×	Usage of key derivation function MGF1 is undefined in protocol. Inconsistent naming of variables SVK and svk.
4.9 SetupTallyCCM$_j$	✓	Parameter μ undefined in protocol. Otherwise, sufficient alignment reached.
4.10 SetupTallyEB	×	Parameter δ, generation of secret key shares, and compressing of public keys undefined in protocol.

Voting Phase

5.1 GetKey	×	Usage of key derivation function MGF1 is undefined in protocol.
5.2 CreateVote	✓	Inconsistent use of symbols between $(\gamma_1, \phi_{1,0})$ and $(c_{1,0}, c_{1,1})$ and between $(\gamma_2, \phi_{2,0}, \ldots, \phi_{2,\psi-1})$ and $(c_{2,0}, \ldots, c_{2,\psi})$. Otherwise, sufficient alignment reached.
5.3 VerifyBallotCCR$_j$	×	List of already accepted valid votes $\mathsf{L}_{\mathtt{validVotes},j}$ with membership tests not implemented. Equality test $\psi = \psi^*$ not implemented.

5.4 PartialDecryptPCC$_j$	×	List of already accepted valid votes $\mathsf{L}_{\texttt{validVotes},j}$ with membership tests not implemented. Equality test $\psi = \psi^*$ not implemented. Specification of $\mathbf{i}_{\texttt{aux}}$ as a nested list unclear. Inconsistent naming of variables $\mathsf{L}_{\texttt{partDec},j}$ and $\mathsf{L}_{\texttt{decPCC},j}$.
5.5 DecryptPCC$_j$	×	$\mathsf{L}_{\texttt{validVotes},j}$, $\mathsf{L}_{\texttt{decPCC},j}$ and $\mathsf{L}_{\texttt{partPCC},j}$ with membership tests not implemented. Missing input values $\phi_{2,0}, \ldots, \phi_{2,\psi-1}$ in Line 2. Invalid range $[0, \psi - 1)$ in Line 14. Specification of $\mathbf{i}_{\texttt{aux}}$ as a nested list unclear.
5.6 CreateLCCShare$_j$	×	Ensure $\mathsf{L}_{\texttt{confirmedVotes},j}\{\texttt{vc}_{\texttt{id}}\} = 0$ requires $\mathsf{L}_{\texttt{confirmedVotes},j}$ to be initialized, but initialization is missing. Protocol defines $\mathsf{L}_{\texttt{decPCC},j}$, $\mathsf{L}_{\texttt{sentVotes},j}$, $\mathsf{L}_{\texttt{confirmed},j}$ to contain ballots $\texttt{b}_{\texttt{id}}$, not voting card identifiers $\texttt{vc}_{\texttt{id}}$. Output $\texttt{K}_{j,\texttt{id}}$ not specified in protocol.
5.7 ExtractCRC	×	Usage of key derivation function MGF1 is undefined in protocol. Updating of $\mathsf{L}_{\texttt{sentVotes},j}$ and $\texttt{bb}$ is missing.
5.8 CreateConfirmMessage	✓	Sufficient alignment reached.
5.9 CreateLVCCShare$_j$	✓	Variable `maxConfAttempts` replaced by constant value 5. Inconsistent naming of variables $\texttt{hcm}_{\texttt{id}}$ and $\texttt{hCK}_{\texttt{id}}$. Output $\texttt{Kc}_{j,\texttt{id}}$ not specified in protocol. Otherwise, sufficient alignment reached.
5.10 ExtractVCC	×	Usage of key derivation function MGF1 is undefined in protocol. Updating of $\mathsf{L}_{\texttt{confirmedVotes},j}$ not implemented.

Tally Phase

Cleansing	×	Pseudo-code algorithm is missing in Section 6.1.1.
6.1 MixDecOnline$_j$	✓	Handling of commitment key $\texttt{ck}_{\texttt{mix}}$ unclear. Special case $\texttt{N}_\texttt{C} = 0$ treated as a special case (not necessary). Inconsistent use of symbols $\overline{\mathbf{c}}_i$ and $\mathbf{c}_{\texttt{dec},j}$. Compression of public keys unspecified in protocol. Otherwise, sufficient alignment reached.
6.2 MixDecOffline	×	Same remarks as for MixDecOnline$_j$. Wrong order of keys $(\texttt{EB}_{\texttt{sk}}, \texttt{EB}_{\texttt{pk}})$ instead of $(\texttt{EB}_{\texttt{pk}}, \texttt{EB}_{\texttt{sk}})$ in Line 6.
6.3 DecodePlaintexts	✓	Sufficient alignment reached.

Verification Algorithms

During the protocol execution, four verification algorithms VerifyConfigPhase, VerifyVotingPhase, VerifyOnlineTally, and VerifyOfflineTally are executed by the auditors. While high-level descriptions of these algorithms are given in [ProtSpec, Sect. 12], correspond-

ing pseudo-code algorithms are unfortunately missing in [SysSpec]. However, they are included in the *Verifier Specification* document [VerifSpec], which we received in early September. Each of the four algorithms defines a *verification block*, which itself consists of multiple sub-algorithms to be called to conduct corresponding verification steps. These sub-algorithms are listed in the following table (some of the sub-algorithms depend on several *supporting algorithms*, which we are not listed in the table below).

Given the late release of this document in the middle of the examination period, we were not able to check the alignment with the protocol in every detail. Generally, we found it quite hard to even establish the links to the verification steps as defined [ProtSpec], especially in the case of VerifyVotingPhase (Block 2), which consists of several relatively complex sub-algorithms. In the case of VerifyConfigPhase (Block 1), we observed that the first two sub-algorithms are not included in the protocol specification.

This observation raises the question of the target audience and purpose of this document, because it seems to contain verification steps that are essential for an independent external auditor, but not necessarily for an internal auditor involved in the protocol. In our Scope 1 report, we have already expressed our concern regarding the extended role of the auditor as an active protocol party, instead of a passive party that acts only in the aftermath of an election. The impression of [VerifSpec] as a document with an unclear target audience confirms this concern.

Block 1: VerifyConfigPhase		
1.01 VerifyEncryptionParameters	×	Not included in protocol specification.
1.02 VerifyVotingOptions	×	Not included in protocol specification. Creates an endless loop if test in Line 5 fails.
1.21 VerifyEncryptedPCCExponentiationProofs	–	Corresponds to Step 2. Alignment not further examined.
1.22 VerifyEncryptedCKExponentiationProofs	–	Corresponds to Step 3. Alignment not further examined.
Block 2: VerifyVotingPhase		
2.21 VerifyPCCExponentiationProofs	?	Difficult to link with protocol.
2.22 VerifyExtractableCC	?	Difficult to link with protocol.
2.23 VerifyCKExponentiationProofs	?	Difficult to link with protocol.
2.24 VerifyNonFinalConfirmationAttempts	?	Difficult to link with protocol.
2.25 VerifyListOfVotesWithSuccessfulConf.	?	Difficult to link with protocol.
2.26 VerifyMixNetInitialPayload	?	Difficult to link with protocol.
Block 3: VerifyOnlineTally		
3.01 VerifyOnlineShuffleProofs	–	Corresponds to Step 1 under "*Verify the CCM's NIZK proofs*". Alignment not further examined.

3.02 VerifyOnlineDecryptionProofs	–	Corresponds to Step 2 under "*Verify the CCM's NIZK proofs*". Alignment not further examined.
Block 4: VerifyOfflineTally		
4.11 VerifyOfflineShuffleProof	–	Corresponds to Step 1 under "*Verify the last CCM's NIZK proofs*". Alignment not further examined.
4.12 VerifyOfflineDecryptionProofs	–	Corresponds to Step 2 under "*Verify the last CCM's NIZK proofs*". Alignment not further examined.
4.13 VerifyPlaintextsDecoding	–	Corresponds to "*Verify decoding*". Alignment not further examined.

2.2 Deviations Between System Specification and Source Code

The main goal of our source code analysis was to check its alignment with the pseudo-code algorithms from the system specification. In our own work [2,3], based on applying strict naming conventions and programming principles, we have shown how far the alignment between pseudo-code and programming code can go in an e-voting project. Our hope upon accepting our involvement in this assessment process was to encounter an alignment level that is comparable to ours. In certain parts of the code, which seem to have undergone a major revision compared to earlier versions, this is actually the case. In those parts, class and methods names can be linked easily to the specification, variable names refer more or less one-to-one to the mathematical notation used in the pseudo-code algorithms, and the order of the executed statements is mostly identical. The overall matching in the revised parts of the code is not perfect, but generally very good.

In other parts of the code, however, quite the opposite is true. There we found many inconsistencies, naming conflicts, and other irritating differences between pseudo-code and programming code. Sometimes, it was even difficult to locate the right code that executes a given algorithm. Given the overall absence of similarities in those parts of the code, checking the alignments required hard work or was simply impossible to do. The fact that we have found two types of code of different quality, one that is relatively clean and well structured and one that is cluttered and confusing, is a clear sign that the source code provided is in a transitional state.

An example for this is the existence of two cryptographic libraries, the original one from Scytl called cryptolib and a revised new one called crypto-primitives. Both libraries are used extensively in the code, sometimes simultaneously by the same method. Currently, there is even an overlap of functionality between the two libraries, which results in code duplication and unnecessary redundancy. We assume that cryptolib is supposed to disappear when all its functionalities have been shifted to crypto-primitives, but unfortunately

this process is not yet completed. In production-ready code, such incomplete transitions are normally not acceptable. In Section 2.3, we will describe further observations regarding the crypto-primitives library.

2.2.1 General Deviations

In the given pseudo-code descriptions, every algorithm has a name, a list of input parameters, a number of code lines to be executed, some local variables, and a well-defined output. The code lines contain common control structures such as if-then-else statements or classical for- or while-loops. In total, we counted 50 algorithms in [SysSpec] and 32 algorithms in [VerifSpec]. Since this is a manageable quantity, we expected to find a corresponding set of Java or JavaScript methods with identical names, inputs, code lines, local variables, and outputs, ideally in more or less the same area of the code base. This is how we understand perfect alignment between specification and code. We already mentioned that certain revised parts of the code have reached a satisfactory alignment level, but many other parts have not.

A major problem that clearly hinders a straight alignment in many places is the use of complex software frameworks such as Spring, which introduces programming concepts such as *batch processing* or *code injections*. While such frameworks are useful in general software development projects for increasing the speed of developing code following certain repeating patterns, they also introduce additional layers of complexity. Generally, arranging the code as intended by such frameworks tends to obscure the overall program flow, which is in conflict with the idea of a simple implementation following the pseudo-code as closely as possible. We have encountered cases, in which the algorithm dissolves completely within the framework, in such way that the task of examining the code alignment gets almost impossible. We therefore strongly recommend not using external frameworks, at least not for implementing the given pseudo-code algorithms of the cryptographic protocol.

We have also observed in some revised parts of the code, especially in the new crypto-primitives library, that Java streams are often used to substitute classical iteration constructs such as for- or while-loops. While Java streams get increasingly popular among developers and are a very powerful new tool, they are not optimal for implementing critical cryptographic algorithms with the goal of optimizing their alignment with the pseudo-code. We do not generally disapprove streams from being used for this purpose, but then we would prefer to see them as part of a consistently applied programming style. At the moment, some iterations are implemented with classical loops, some with stream pipelines, and some with batch processing techniques. This diversity of technologies for essentially the same purpose makes the examination unnecessarily challenging. Here a clear and consistent concept seems to be missing.

Another general problem is the existence of code that seems vital for the implementation of the cryptographic protocol, but for which no pseudo-code exists in [SysSpec]. A

prominent example of this type is the cleansing procedure, for which a vast amount of source code exists, but without any matching pseudo-code. In such areas of the code, checking the alignment is inherently impossible. Another such example can be found in the client code for submitting a ballot. According to the algorithm CreateVote in both the protocol and the system specification, ballots are submitted unsigned, but in the implementation, we found code for creating a signature and attaching it to the ballot. In perfectly aligned code, protocol messages contain exactly the information as specified, not more and not less.

2.2.2 Deviations Between Algorithms

In our tabular overview given below, we provide a distinction of different cases of misalignment between the pseudo-code and programming code algorithms. An implementation diverging from the pseudo-code is marked with a cross (×). An implementation where the alignment to the pseudo-code is very hard to justify or even impossible is marked with a question mark (?). This can happen due to naming conflicts or if the implementation is scattered over multiple classes and methods. The case, where both types of misalignment are present simultaneously is marked with both a cross and a question mark (×?).

Configuration Phase

4.1 GenKeysCCR$_j$	×?	The specification suggests a very short implementation, however it is quite difficult to find it in the code. k'_j is not generated using `GenRandomIntegerUpperBound`, but as an ElGamal key pair.
4.2 GenVerCardSetKeys	×	Almost impossible to find, as it is in no correlation with the specification. Neither does the name of the class reflect the specification, nor its content.
4.3 GenSetupEncryptionKeys	✓	Sufficient alignment reached.
4.4 GenVerDat	×	There is no alignment to L_{pCC}.
4.5 GenEncLongCodeShares$_j$	✓	Sufficient alignment reached.
4.6 CombineEncLongCodeShares	?	Implementation as batch-job breaks for-loop alignment with specification.
4.7 GenCMTable	×?	Implementation as batch-job. Almost impossible to map to the specification. CMtable is created per voter, whereas specification states one CMtable for all voters combined.

4.8 GenCredDat	×?	Impossible to map to the specification. It boils down to the following comment that can be found in the class `ConfigJobConfig`: *"This step creates and stores in a 'cache' bean some extra complex classes needed for the voting card generation that are not possible (or easy) to store in the standard spring batch job execution context."*
4.9 SetupTallyCCM$_j$	×?	The implementation does not seem to follow any paradigm that allows any mapping to the pseudo-code of the specification. Here, the bypassing of the `crypto-primitives` library is decorated with the following comment: *"This method is equivalent to the crypto-primitives' GenKeyPair method."*
4.10 SetupTallyEB	×?	The implementation is not aligned with the three lines of pseudo-code within the specification. The algorithm has been ripped apart and embedded in multiple methods. So it is very hard to find any correlation.

Voting Phase

5.1 GetKey	?	The implementation of this algorithm is completely teared apart. Algorithm internal values are stored in the session and persisted over time.
5.2 CreateVote	×	The proof does not include $\mathbf{i}_{\mathtt{aux}}$ at all. The ballot is signed and the signature as well as the certificate are sent to the voting server. This is unspecified and in fact contradicts the specification which states that the voter is only *"implicitly authenticated"* [SysSpec, Sect. 7].
5.3 VerifyBallotCCR$_j$	×?	The input values for the proofs and the hash are wrong and the required $\mathbf{i}_{\mathtt{aux}}$ is missing. The implementation is in a very bad shape, and does not provide any helping hand in mapping back to the specification.
5.4 PartialDecryptPCC$_j$	×?	The input values for the proofs and the hash are wrong and the required $\mathbf{i}_{\mathtt{aux}}$ is missing. The implementation is in a very bad shape, and does not provide any helping hand in mapping back to the specification. Here, 5 lines of pseudo-code have been dissolved to more than 100 lines of implementation code.
5.5 DecryptPCC$_j$	×?	In the implementation this algorithm is executed by the voting Server and not by the control components as specified. The variables used are not aligned with the specification.

5.6 CreateLCCShare$_j$	×?	Important membership checks involving L_{decPCC} and $L_{sentVots}$ are missing. The generated proof is not aligned.
5.7 ExtractCRC	×?	The CMtable is per voter. This is not aligned with the specification where there is only one CMtable for all voters.
5.8 CreateConfirmMessage	✓	Sufficient alignment reached.
5.9 CreateLVCCShare$_j$	×?	Same situation as for CreateLCCShare$_j$.
5.10 ExtractVCC	×	Same situation as for ExtractCRC.

Tally Phase

Cleansing	×?	Even though no specification exists for cleansing, we tried to locate the cleansing algorithm in the implementation. However, we were not able to find the corresponding code block, not even with some help from Swiss Post. This provides strong evidence that the use of a complex framework, such as Spring, negatively interferes with auditability.
6.1 MixDecOnline$_j$	×	The list L_{bb} is not reflected in the implementation. Thus there is also no check whether bb has already been mixed and decrypted.
6.2 MixDecOffline	✓	The values ee and bb are passed as input parameters and not taken from context as specified. Specification should be adjusted. Otherwise, sufficient alignment reached.
6.3 DecodePlaintexts	×	Decoding missing in the implementation.

2.3 Crypto-Primitives

The library crypto-primitives, which has been released as a separate and independent project, is mostly in a very good state. Most of the code can be mapped very easily to corresponding pseudo-code algorithms in the given specification document [CryptoPrim]. We encountered a number of minor deviations and inaccuracies, but none of them are critical. The level of alignment in this part of the system is therefore already sufficiently high in most parts.

Based on our observations, we can make some recommendations to improve the quality of the library even further. The first recommendations concern the pseudo-code algorithms in [CryptoPrim]:

- Many algorithm descriptions have a section called **Ensure** with preconditions relative to the input variables. Other algorithm have a similar section called **Require**.

In other cases, preconditions are listed along with the input parameters, and there are even cases where preconditions are tested explicitly in the first lines of the algorithm. The difference between these four types of preconditions is currently not clear, and this gives the impression that a proper concept for handling preconditions is currently missing. We recommend adding such a concept to the document and applying it strictly to all algorithms.

- The two algorithm parameters n and m of the shuffle proof by Bayer and Groth can be chosen freely under the constraint $N = nm$. We have seen that the selection of n and m is implemented in a way that optimizes the size of the proof, but at the same time maximizes the computational costs (by choosing n and m closest to $\sqrt{N}$). We think that optimizing the proof size is not really necessary, i.e., we would recommend solving this trade-off in favor of the computational costs, simply by setting $m = 1$ and $n = N$. Another benefit of this particular choice is that it renders large parts of the proof obsolete, i.e., ZeroArgument, HadamardArgument, and productArgument from [CryptoPrim, Sect. Fig. 1] could be dropped entirely. This would greatly help to simplify the most complex parts of both the document and the code.
- In the proof generation and verification algorithms of [CryptoPrim, Sect. 6], the array of auxiliary information strings $\mathbf{i}_{\mathsf{aux}}$ is always added to $\mathbf{h}_{\mathsf{aux}}$ as an additional element. This creates a nested structure of elements to be hashed. In the implementation of these algorithms, we observed that the values from $\mathbf{i}_{\mathsf{aux}}$ are added to $\mathbf{h}_{\mathsf{aux}}$ using the method `concat` from the `Streams` class. This is therefore a deviation between specification and implementation, which should be eliminated. We recommend using concatenation also in the pseudo-code, which then permits to remove the repeating remarks "If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it".

We have also some remarks that may help to further improve the Java code of the library:

- The purpose of the interface `CryptoPrimitives` is not clear. It has the comment "*Interface exposing all methods that need to be accessed outside of crypto-primitives*", but the interface contains only four methods related to generating random strings and random integers. The only class implementing this interface is `CryptoPrimitivesService`, which is mainly a wrapper class for `RandomService`. It thus seems that the interface and the class are not really useful.
- We observed that some Google libraries are used accross crypto-primitives, for example for checking preconditions, for Base32 or Base64 encodings, or for immutable lists. Most of the imported functionalities are relatively simple, i.e., it would not be difficult to provide them directly. Generally, we recommend reducing the dependencies to third-party libraries as much as possible, especially in cases where the benefit of the imported library is marginal.
- Most pseudo-code algorithms depend on a context, which usually consists of several global parameters such as the group parameters p and q. However, no such context

exists in the implementation, i.e., global parameters are provided implicitly, for example by selecting the group from an input element and the generator from that group using statements like `pk.getGroup().getGenerator()`. We recommend to introduce an explicit context object, which is passed to all methods as an additional parameter. The actual input elements can then be checked against this context.

- The algorithms are spread over a large number of classes. Many of these classes define instance variables, which are then implicit parameters for corresponding methods. We see this as a conflict between the object-oriented programming paradigm in Java and the procedural programming paradigm in the pseudo-code algorithms, which makes a one-to-one alignment more difficult. We would recommend to define the algorithms as static methods which operate on pure data objects (an exception is the algorithm GetCiphertextVectorExponentiation, which is already implemented as a static method).
- Some algorithm names are different in the code: GetCiphertextExponentiation is implemented as a method called `exponentiate`, and GetCiphertextProduct is implemented as a method called `multiply`. We recommend applying strict naming conventions everywhere.

2.4 Underspecified Concepts and System Components

The current implementation includes some components that are relevant for the cryptographic protocol, but which are not sufficiently well specified. In some cases, the problem seems to come from missing or unclear underlying concepts. For all the topics discussed below, we recommend that further explanations are given in the system specification.

- In the protocol specification, the states of the involved parties are called *Logs*. Corresponding objects $\mathsf{Logs}_{\mathsf{PO}}$, $\mathsf{Logs}_{\mathsf{CCR},j}$, and $\mathsf{Logs}_{\mathsf{CCM},j}$ appear everywhere in the algorithms for collecting the received messages and the computed results of the respective parties. In our Scope 1 report, we have already remarked that the exact content of these objects is underspecified and that their purpose and structure should be defined more accurately. Given the significance of the logs in [ProtSpec], we were surprised to observe that the logs are not included in pseudo-code algorithms and that they are not even mentioned in the whole specification document (they are discussed in a separate section of [VerifSpec], but this is a completely different context). On the other side, we observed that objects of type `SecureLog` are included in the implementations of the CCR and verifier components, but not in the CCM and print office components. Unfortunately, no information is given to explain this inconsistency.
- Many of the involved components, especially the four control components, depend on various data structures such as three different lists of voting cards $\mathsf{L}_{\mathtt{decPCC}}$, $\mathsf{L}_{\mathtt{sentVotes}}$, and $\mathsf{L}_{\mathtt{confirmedVotes}}$, the so-called "partial choice return codes allow list"

$\mathsf{L}_{\texttt{pCC}}$, and the list of shuffled and decrypted ballot boxes $\mathsf{L}_{\texttt{BB}}$. These lists are important for keeping track of previous protocol events. Other such data structures are the "return codes mapping table" CMtable and the "primes mapping table" pTable. The exact shape and the initialization of these data structures is not always very clear, for example $\mathsf{pTable} = (\tilde{\mathbf{p}}, \tilde{\mathbf{v}})$ seems to be a pair of two vectors rather than a (two-dimensional) table. Sometimes, these lists and tables are treated as sets, for example by performing membership tests such as $\texttt{vc}_{\texttt{id}} \in \mathsf{L}_{\texttt{decPCC},j}$ or set operations such as $\mathsf{L}_{\texttt{sentVotes},j} \leftarrow \mathsf{L}_{\texttt{sentVotes},j} \cup \texttt{vc}_{\texttt{id}}$ (with missing curly brackets) in algorithm $\mathsf{CreateLCCShare}_{\mathsf{j}}$. We recommend a stricter use of common mathematical notation in these examples. Generally, it would be good to introduce different types of data structures (list, sets, tables, maps) in one place of the specification, together with a discussion of their initialization, permitted operations and general properties.

- At first sight, the descriptions of the **correctnessID** concept seem to correspond in [ProtSpec, Sect. 10.4] and in [SysSpec, Sect. 3.4.4], except that [SysSpec] contains two trivial algorithms for selecting values from corresponding vectors. In our Scope 1 report, we already stated that there must be two such vectors, even if this is not evident given the description from [ProtSpec]. In addition to resolving the current misalignment between the two documents, we recommend to better specify the purpose, properties, and initialization of these vectors. The given description only show some exemplary values like "aaaaa1" or "bbbbb2", but it does not clearly specify how to select them in an actual implementation. We found an implementation of this topic in the Class `CombinedCorrectnessInformation`, but linking it to the current description in [SysSpec] is anything but obvious.
- Algorithm GenKeyPair as defined in [CryptoPrim] generates a vector of key pairs of size N, i.e., $\texttt{EB}_{\texttt{sk}}$ in Algorithm SetupTallyPO is a vector of secret keys of size δ. This is in conflict with the input parameter s of SplitSecretShares, which expects a single secret value. It is therefore unclear how the encoding of $\texttt{EB}_{\texttt{sk}}$ into a single secret value s works, such that SplitSecretShares can create corresponding shares using Shamir's secret sharing scheme. It is also not clear how the modulo $r > s$ of the prime field $\mathbb{Z}_r$ is chosen in SplitSecretShares, especially because s will have the size of approximately $\delta \cdot |q|$ bits. These aspects must be better specified.
- A question that we raised already in Scope 1 is how the voters receive their respective key stores and other election data. To the best of our understanding, voters are never asked to enter their voting card identifier $\texttt{vc}_{\texttt{id}}$, but without entering $\texttt{vc}_{\texttt{id}}$, selecting the correct key store seems impossible. We expected to find better explanations in [SysSpec], but this was unfortunately not the case. Without spending too much effort, we were also not able to quickly answer this question by looking at the source code. We recommend including a clarifying discussion of this topic in the specification document.
- The cryptographic protocol uses authenticated symmetric encryption at several places. The discussion and pseudo-code algorithms in [SysSpec, Sect. 8.4] delegate the encryption and authentication tag generation to corresponding sub-

algorithms, and the same holds for the decryption and tag verification. Unfortunately, these sub-algorithms are not further specified. In the comments given on that topic, "*the existence of an authenticated symmetric encryption and decryption function, such as AES-GCM 128*", is assumed, but it is not clear if this particular choice is mandatory for an actual implementation. The comment itself is somewhat confusing, since AES-GCM generates the ciphertext and the authentication tag simultaneously, whereas the two top-level algorithms GenCiphertextSymmetric and GetMessageSymmetric process them separately. In the class `ConfigSymmetricCipherAlgorithmAndSpec` from the cryptolib library, we found an enum constant `AES_WITH_GCM_AND_NOPADDING_96_128_BC`, which indicates that the AES-GCM implementation from BouncyCastle library is actually used. However, since the cryptolib has complex APIs and code that is difficult to understand and analyze, we cannot confirm that AES-GCM is properly implemented. With respect to symmetric encryption, we recommend improving both the level of details given in the specification and the clarity of the implementation, with the goal of aligning to as much as possible.

2.5 Quality of Code

For assessing the code quality of a software system, there are standardized measures for evaluating certain quality aspects—for example the degree of code duplication or the entanglement of components—and there are various great tools on the market for deriving these measures automatically from the code base. Knowing that others have already analyzed the Swiss Post system using such tools, we decided not to repeat this task. As an example of the results found by others, we depict below the summary from applying the TÜViT/SIG *Model for Software Product Maintainability* to the Swiss Post system. This example is taken from the *Technical Report – Auditability Assessment E-Voting* (August 2021) by Kay Grosskop and Marc A. Hahn from sieber&partners, which is available on the Swiss Post web page. As one can see, the system has mostly received average or slightly above average marks for most evaluation criteria.[1]

Volume	★★★★☆	4.1
Duplication	★★★☆☆	2.5
Unit size	★★★☆☆	3.2
Unit complexity	★★★★☆	4.0
Unit interfacing	★★☆☆☆	2.3
Module coupling	★★★★☆	3.3
Component balance	★★★★☆	4.4
Component independence	★★★★☆	3.6
Component entanglement	★★☆☆☆	2.4

[1] In the same report, other product quality criteria such as *readability* and *platform choice* have been assessed. In those topics, which we consider less relevant from a security perspective, the system received relatively good marks.

There are also very useful tools for static and dynamic code analysis, which can be used to locate all kind of problems in the code, from simple formatting inconsistencies to high-risk vulnerabilities. Other tools exist for finding performance issues or for determining the test and documentation quality. Knowing that Swiss Post applies such tools routinely in their development process, we decided not use them in our assessment.

Our contribution to the evaluation of the system's code quality consists of a discussion of certain problematical topics that we encountered while looking at the code. Each discussed topic includes recommendations for possible improvements.

- In a cryptographic protocol, it is important that the information contained in the exchanged messages is exactly of the expected form and consistent with given parameters from the current context. To guarantee this property in an actual implementations, messages received from another party must be checked rigorously. A clear input validation concept is required to define the necessary checks efficiently and to implement them consistently in all parts of the protocol. Currently, such a clear concept seems to be missing. For example, we observed cases in which more or less the same tests are repeated multiple times for the same data. Submitted votes, for example, are parsed and checked twice by the CCRs. Generally, the code for performing these steps is very complex and sometimes quite confusing, as one can easily see for example by inspecting the class `PartialChoiceReturnCodesDecryptionConsumer`, which belongs to the implementation of the algorithm $\mathsf{VerifyBallotCCR_j}$. Auditing such low-quality code is almost impossible.
- In Section 2.2, we already mentioned the existence of two similar cryptographic libraries with partly overlapping functionalities. This overlap implies several code quality problems such as dead or redundant code. In some parts of the implementation, where both libraries are used simultaneously, we found rather confusing code segments. The class `CleansedBallotBoxService`, which deals with ElGamal encryptions using both libraries, is an illustrative example. The following import statements of this class show the strong dependencies to both libraries. Among the imports, three classes represent essentially the same mathematical concept, `ElGamalPublicKey` and `ZpGroupElement` from cryptolib and `GqElement` from crypto-primitives. From a software engineering perspective, the existence of multiple units for the same concept could be called a violation of the cohesion principle.

```
import ch.post.it.evoting.cryptolib.api.asymmetric.AsymmetricServiceAPI;
import ch.post.it.evoting.cryptolib.api.exceptions.GeneralCryptoLibException;
import ch.post.it.evoting.cryptolib.certificates.utils.CryptographicOperation
       Exception;
import ch.post.it.evoting.cryptolib.elgamal.bean.ElGamalPublicKey;
import ch.post.it.evoting.cryptolib.mathematical.groups.impl.ZpGroupElement;
import ch.post.it.evoting.cryptoprimitives.elgamal.ElGamalMultiRecipient
       Ciphertext;
import ch.post.it.evoting.cryptoprimitives.elgamal.ElGamalMultiRecipient
       PublicKey;
import ch.post.it.evoting.cryptoprimitives.hashing.HashService;
```

```
import ch.post.it.evoting.cryptoprimitives.hashing.HashableList;
import ch.post.it.evoting.cryptoprimitives.math.GqElement;
import ch.post.it.evoting.cryptoprimitives.math.GqGroup;
```

- The naming of classes, methods, and variables is not always optimal for achieving self-explanatory and easy-to-read code. An illustrative example is the lengthy method name `combineChoiceCodeNodesDecryptionContributions` from the service class `CodesDecrypterService`. Here, the intention of choosing a most accurate method name results in quite the opposite, completely unreadable code lines whenever this method is called. A few other examples of that kind are the following:
  - class `PartialChoiceReturnCodesDecryptionConsumer`,
  - method `getPartialChoiceReturnCodesVerificationInput`,
  - variable `partialChoiceReturnCodesOrConfirmationKey`.

  Given these examples, it is evident that reading code lines containing such lengthy names and mapping them to the pseudo-code becomes very difficult. To avoid this problem, we recommend to improve existing naming conventions and to strictly apply the revised conventions to all areas of the code.
- In Section 2.2, we already mentioned the code readability and pseudo-code alignment problems that result from using frameworks such as Spring. Especially the use of batch jobs leads to code that is much harder to inspect compared to regular code, especially for people without profound understanding of the Spring Batch technology. To us at least, using this technology obscures the program flow considerably in many critical parts of the system and therefore diminishes the overall code quality of the system. An example of a batch job implementation is the algorithm GenCredDat. The salt used to derive the keystore key $\texttt{KSkey}_{\texttt{id}}$ from the start voting key $\texttt{SVK}_{\texttt{id}}$ is not created in the implementation of the algorithm itself, but taken from the job execution context, which is pre-filled by a tasklet in a preliminary step controlled by Spring Batch. This example demonstrates how simplicity gets lost without any apparent benefit. Therefore, we recommend to avoid these technologies at least in those areas of the code that are relevant for the cryptographic protocol.
- A general code quality aspect is the strict inclusion of up-to-date libraries. This is currently not the case. The most critical example is the dependency to the web framework AngularJS from Google. As Long Term Support (LTS) for AngularJS ends on December 31, 2021, Google will no longer fix security or browser compatibility issues. Updating the code to the new Angular framework should therefore not be postponed any further. The update has been announced in the project repository's readme.md file, but it still an open issue. We generally recommend using the latest updates of all external libraries. The following list of Java libraries that are outdated for more than one year is taken from the above-mentioned technical report by sieber&partners. Note that some dependencies are outdated for almost 10

years. Generally, we also recommend moving towards a newer LTS Java version, such as the latest Java 17 from September 2021. Support for the currently used Java 1.8 is expected to run out in the near future.

Examples of libraries that have a new version for more than one year*

Dependency	Current version	Current version date	Newer version	Outdated since
org.jboss.arquillian:arquillian-bom	1.4.0.Final	27.02.18	1.6.0.Final	19.10.18
org.eu.ingwar.tools:arquillian-suite-extension	1.1.4	11.04.17	1.2.2	15.03.18
org.apache.xmlgraphics:batik-css	1.1	11.05.18	1.14	01.02.19
commons-codec:commons-codec	1.9	21.12.13	1.15	06.11.14
org.hibernate:hibernate-core	4.3.8.Final	06.01.15	5.5.3.Final	15.04.15
org.hibernate:hibernate-core	3.6.10.Final	09.02.12	5.5.3.Final	15.12.11
org.hibernate:hibernate-entitymanager	4.3.8.Final	06.01.15	5.5.3.Final	15.04.15
org.hibernate:hibernate-entitymanager	3.6.10.Final	09.02.12	5.5.3.Final	15.12.11
org.hibernate:hibernate-validator	4.2.0.Final	20.06.11	7.0.1.Final	09.05.12
javax.xml.bind:jaxb-api	02.02.11	09.2013	02.03.01	10.2018
com.sun.xml.bind:jaxb-core	2.2.11	10.10.14	3.0.1	02.08.17
com.sun.xml.bind:jaxb-impl	2.2.11	10.10.14	3.0.1	02.08.17

2.6 Synchronization

In implementations of cryptographic protocols with multiple parties, there is no guarantee that the messages are exactly exchanged as defined in the protocol, especially if the adversary model includes active adversaries who can deviate from the protocol in any possible way. Therefore, messages can be blocked, delayed, modified, or replayed, depending on the adversary's attack strategy. Blocked or delayed messages caused by network failures also exist in the absence of an adversary. In any case, a party expecting certain protocol messages must be aware that the chronological order of the messages may have been altered during transport, that copies of the same message may arrive at different times, and even that different instances of the same type of message may arrive from the same party. In the realization of such a party as a software component, it is therefore important to consider these possibilities and to implement a robust strategy for sorting out the incoming messages, the ones to be kept and the ones to be thrown away.

The most critical situation of that kind arises when two messages of the same type and from the same sender arrive at almost exactly the same time, for example two different ballots from the same voter. While ordinary voters using the provided web interface of the official election portal will normally not be able to submit more than one ballot simultaneously, one should assume that such situations can be provoked by an adversary, who might want to learn the choice return codes for multiple ballots. If such a situation occurs, it is therefore critical to avoid that more than one ballot is processed by the voting system. In the current implementation of the Swiss Post system, three lists

$L_{\texttt{decPCC}}$, $L_{\texttt{sentVotes}}$, and $L_{\texttt{confirmedVotes}}$ are managed by the control components for keeping track of the submitted votes and confirmations. These lists are therefore used to decide whether an incoming ballot or confirmation should be processed or not. In Section 2.4, we have already criticized that as general discussion of the initialization, permitted operations, and general properties of these lists is missing in the documentation. On particular missing aspect in both the documentation and the implementation is the synchronization of these lists in an environment where they are used simultaneously in parallel processes. Without applying proper synchronization measures, it is possible that the mutual exclusion of processing two simultaneous messages of the same voter is no longer guaranteed. As explained above, this may then undermine the verification properties of the underlying cryptographic protocol.

Synchronization problems of that kind are well known in general web applications with a large number of simultaneous users, and methods to prevent them are well-known and not difficult to implement. In the current implementation, we were therefore surprised not to find the necessary synchronization measures that we would have expected to see for protecting the integrity and consistency of the above lists. We discussed this topic with some developers from Swiss Post in on online meeting, in which they confirmed that currently no synchronization measures exist.

2.7 Randomness

As for the generation of high-quality randomness, the implementation uses the standard cryptographic pseudo-random generators (PRG) provided by the respective programming languages. This is realized in Java using an instance of `SecureRandom` and in JavaScript using the function `getRandomValues` from the Web Crypto API. In both cases, using the given standard components is a legitimate decision to ensure that the respective PRG implementations are correct with high probability. The general problem with pseudo-randomness is to provide a high-entropy seed for the PRG initialization and later for the re-seeding of the PRG at regular intervals. To obtain a high-entropy seed when needed, the quality of the entropy source must be guaranteed at all times. Given the importance of a reliable PRG for cryptographic applications, the entropy source is therefore one of the most critical components. If an attacker succeeds in controlling the entropy source, then the properties of the cryptographic techniques in use are no longer guaranteed. An additional problem is the fact that attacks against the entropy source leave almost no traces and are therefore difficult to detect.

In Java, without taking further actions, `SecureRandom` obtains the seed directly over the operating system's entropy API. This means that the security of the application is delegated entirely to the administrator of the machine on which the application is running. In Unix-like operating systems, the following code snippet is sufficient for redirecting the entropy source to return a sequence of 0's, instead of returning high-entropy values collected from the system's environmental noise. It only serves as an example here to demonstrate the delicacy of the issue:

```
ln -s -force /dev/zero /dev/urandom.
```

From our point of view, it is not legitimate to assume that no adversary manages to control the entropy API of the machines running the voting system. Attacks of that kind are too simple and their consequences are devastating.

To solve this problem, we recommend to complement the seeds obtained from the default entropy source by entropy from other sources. Note that complementing seeding with extra entropy from other sources can never decrease the quality or security of the generated pseudo-randomness. If options exist for complementing the default entropy source, we therefore recommend exploiting them independently of their quality. One possibility is to extract entropy from so-called *CPU execution time jitter* [4] or from unpredictable concurrency effects [1]. Both techniques can be implemented as software-only solutions that work on any machine. For such an implementation, we recommend following the techniques proposed in the NIST Special Publication SP.800-90B, which includes methods for performing so-called *health tests*. The purpose of health tests is to detect deviations from the intended behavior of the source with high probability and as quickly as possible. In the attack scenario given above, a suitable health test would immediately detect the re-directed entropy source that always returns 0. An exemplary implementation of such methods exists in [3].

In the given Java implementation, we observed that multiple instances of `SecureRandom` are used simultaneously at different places in the code, but mostly in corresponding classes of the two cryptographic libraries cryptolib (for example in `SecureRandomFactory`, `CryptoRandomInteger`, or `CryptoRandomString`) and crypto-primitives (for example in `RandomService` or `GqGroupGenerator`). Note that each instance receives its own seed from the default entropy source. Given the importance of this fundamental topic, we recommend limiting the responsibility of generating randomness to one single class, which is then responsible for providing a sufficient amount of reliable entropy sources, and which is used everywhere for generating cryptographically secure randomness.

At the client side, the most obvious attack vector results from the inclusion of numerous external JavaScript libraries. Even though these libraries are closely observed for known vulnerabilities, they are not tested against targeted overriding of cryptographic functions used in other applications. An attack could therefore be as simple as a single line of code located in a hidden place of an external library, which modifies the behavior of the function `getRandomValues` from the Web Crypto API. For example, if the following code snippet is injected into the voting client by an external library, then `getRandomValues` will output a sequences of 0's instead of the expected high-quality randomness, and this is equivalent to submitting the vote in cleartext. Again, the code snippet only serves as an example to demonstrate the delicacy of the issue:

```
window.crypto.getRandomValues=a=>a.fill(0).
```

As a counter-measure against attacks of that kind, we recommend combining different entropy sources and keeping a reference to the original `getRandomValues` function of the Web Crypto API in a private scope. Additionally, we recommend executing the cryptographic core of the voting client within an `iFrame`-environment, in which no external libraries are allowed.

We are aware that on the client-side, an attack against the randomness source only affects vote privacy, and that in the given adversary model, the voting client is trustworthy against privacy attacks. However, we still think that simple attacks, like the one discussed above, should be avoided to protect the system from being discredited, for example by exposing all submitted votes in cleartext.

References

[1] P. Blanchard, R. Guerraoui, and J. Stainer. Concurrency as a random number generator. Technical Report 215956, EPFL, Switzerland, 2016.

[2] R. Haenni, E. Dubuis, R. E. Koenig, and P. Locher. CHVote: Sixteen best practices and lessons learned. In R. Krimmer, M. Volkamer, V. Cortier, R. Goré, M. Hapsara, U. Serdült, and D. Duenas-Cid, editors, *E-Vote-ID'20, 5th International Joint Conference on Electronic Voting*, LNCS 12455, pages 95–111, Bregenz, Austria, 2020.

[3] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote protocol specification – version 3.2. *IACR Cryptology ePrint Archive*, 2017/325, 2020.

[4] S. H. Müller. CPU time jitter based non-physical true random number generator. Technical report, atsec information security, 2014.