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

Re-Examination of the Swiss Post Internet Voting System

Scope 1 "Cryptographic Protocol" and Scope 2 "Software"

Version 1.0.2

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

February 23, 2023

On behalf of the Federal Chancellery

Revision History

Revision	Date	Description
0.1	12.07.2022	Document initialization.
0.2	15.09.2022	Draft with Section 3 finished submitted to Federal Chancellery.
0.3	04.10.2022	Final draft with all sections finished submitted to Federal Chancellery.
0.4	13.11.2022	Revised final draft with first addendum submitted to Federal Chancellery.
0.5	16.01.2023	Revised draft with second addendum submitted to Federal Chancellery.
1.0	20.01.2023	Final version submitted to Federal Chancellery.
1.0.1	17.02.2023	Updated final version with clarifications added to Addendum-2.
1.0.2	23.02.2023	Updated final version with further clarifications added to Addendum-2.

Contents

Management Summary

This report is the main outcome of our assessment of the Swiss Post e-voting system, which we conducted in the period between June 2022 and January 2023. It is a continuation of a similar assessment conducted one year before on earlier versions of both the cryptographic protocol and the implemented system. The versions that were available for the first assessment were clearly not yet ready for allowing the system to be used in practical elections. In our reports, we listed the encountered problems and recommended corresponding improvements. Generally, we had the impression of looking at an unfinished project that was still heavily under construction.

In the meantime, numerous advancements have been made in many areas of both the cryptographic protocol and its implementation. For example, we observed that the alignment between pseudocode algorithms and source code has been increased significantly, that redundancies in the documentation and inherited problems from the earlier Scytl system have been eliminated, and that the software structure and organization have been improved substantially in many parts, which now makes the code base much more accessible for inspections. We were also glad to see that important technology updates have been made both on the server and the client side, and that the cryptographic part of the implementation has been disentangled from certain non-compulsory dependencies.

Given this broad range of improvements, the subject of our second assessment has been very different compared to the previous one. This allowed us to conduct our examination in much greater detail. For example, we were now able to check the implementation of the cryptographic algorithms on an almost line-by-line basis with the specification. We were also able to look more profoundly into code quality aspects and to examine the general software architecture. Our efforts were driven by the objective of making constructive suggestions for improvements and simplifications.

Our assessment was divided into three evaluation rounds, roughly from June to September 2022, October to November 2022, and December 2022 to January 2023. Between each round, we received a major update of both the specification and the code, in which some of the findings of the previous rounds had already been taken into account. Keeping track of all the changes made in each new release, while keeping a focus on the big picture, was a great challenge during our mission. The structure of this report, with a detailed discussion of the first evaluation round in the main part and corresponding discussions of the received updates in two addendum sections, reflects the difficulties of this process.

Despite the numerous improvements made to the system that we inspected in 2021, our general impression of looking at an unfinished project has not changed much. The new releases that we received between October and December still contained a large number of substantial changes at various places across the whole system, including the documentation. From a system that is close to reaching an important milestone in the project roadmap, we would expect to observe only minor last-minute changes, but

in the present case, quite the opposite was true. The re-introduction of the write-ins functionality in the November release is an example of a complex and subtle topic that carries the risk undesired security problems.

Instead of adding new enhancements to the system, we would have preferred to see further improvements based on our findings and recommendations from the first evaluation round. Certain critical topics, for example the possibility of attacks against the entropy source, have still not been addressed properly. This particular problem was already a major concern in our report from last year, and we repeated our warnings in the new report about the first evaluation round. To us a least, the status of the current system that is available in January 2023 is therefore quite unsatisfactory: some of our findings have been addressed, some have not been addressed, and some new findings came up only with the latest releases shortly before finishing this report.

Given the volatility that is still present in the current system, it is difficult to draw a conclusive verdict about it. However, in the light of the findings listed in this report, we do not think that the system already fulfills all the requirements from the legal ordinance. Without further improvements, it should therefore not be approved for legally binding elections, except possibly for testing purposes under the umbrella of a limited electorate.

1. Introduction

This examination report lists the findings of our 2022 re-assessment of the Swiss Post e-voting protocol and its implementation. We have conducted essentially the same type of work last year for the protocol and system versions that were available in July 2021. Our final reports on Scope 1 (cryptographic protocol) and Scope 2 (software) were delivered to the Federal Chancellery (FCh) in March 2022. Both documents were published on the FCh's web page on April 20, 2022, together with the reports from other experts.[1] In the meantime, the Swiss Post has implemented numerous improvements, which mostly address the findings discussed in the last year's reports.

We have been assigned with this re-assessment task of by the Federal Chancellery in June 2022 for a period of approximately seven months. We submitted a first draft of our report on September 15, 2022, to the Federal Chancellery, a second draft on October 4, 2022, a third draft on November 13, 2022, and the final draft on January 15th, 2023. This stepwise procedure allowed us to clarify and finalize the document based on the feedback that we already received from both the Federal Chancellery and the Swiss Post. And it allowed us also to review the latest updates of the specification and the code, in which certain of our findings and recommendations from the first, second, and third draft have been addressed already. This part of our evaluation is described in two separate addendum sections of this document. In contrast to our last year's reports, we agreed with the FCh to include our Scope 1 and Scope 2 findings in a single report.

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 both the Federal Chancellery and the Swiss Post, mainly for obtaining clarifying information on certain topics. At the beginning of our mission, on June 26, 2022, we were invited to an informal visit of the Swiss Post development team in Neuchâtel. On that occasion, we were given a summary of the current development process and the opportunity to discuss some of the major changes and improvements compared to the last year's version. The participating developers and project managers were all remarkably eager and open to share with us their views of the project and to discuss all sorts of technical questions. This informal visit thus turned into a perfect kick-off meeting for our assessment.

1.1. Relevant Documents

To conduct our assessment, we had to consider the new version of the *Federal Chancellery Ordinance on Electronic Voting* (OEV) and two related documents, its annex and an explanatory report, which contains additional clarifying information. From the Swiss Post, we received five relevant specification documents, which are all publicly available

[1] See https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html

in various repositories of the swisspost-evoting group on gitlab.com.[2] This is therefore the list of relevant documents for this report:

- [OEV] *Federal Chancellery Ordinance on Electronic Voting*, Federal Chancellery FCh, July 1, 2022
- [OEV Annex] *Federal Chancellery Ordinance on Electronic Voting – Annex on Technical and Administrative Requirements for Electronic Voting*, Federal Chancellery FCh, July 1, 2022
- [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 its entry into force on 1 July 2022*, Federal Chancellery FCh, July 1, 2022
- [CryptPrim] *Cryptographic Primitives of the Swiss Post Voting System – Pseudocode Specification*, Version 1.0.0, Swiss Post Ltd., June 24, 2022
  ⇒ Updated to Version 1.0.1 on October 3, 2022 (see Appendix A)
  ⇒ Updated to Version 1.1.0 on October 31, 2022
  ⇒ Updated to Version 1.2.0 on December 9, 2022 (see Appendix B)
- [SysSpec] *Swiss Post Voting System – System Specification*, Version 1.0.0, Swiss Post Ltd., June 24, 2022
  ⇒ Updated to Version 1.1.0 on October 3, 2022 (see Appendix A)
  ⇒ Updated to Version 1.1.1 on October 31, 2022
  ⇒ Updated to Version 1.2.0 on December 9, 2022 (see Appendix B)
- [VerSpec] *Swiss Post Voting System – Verifier Specification*, Version 1.0.1, Swiss Post Ltd., August 19, 2022
  ⇒ Updated to Version 1.1.0 on October 3, 2022 (see Appendix A)
  ⇒ Updated to Version 1.2.0 on October 31, 2022
  ⇒ Updated to Version 1.3.0 on December 9, 2022 (see Appendix B)
- [ProtSpec] *Protocol of the Swiss Post Voting System – Computational Proof of Complete Verifiability and Privacy*, Version 1.0.0, Swiss Post Ltd., July 29, 2022
  ⇒ Updated to Version 1.1.0 on December 9, 2022 (see Appendix B)
- [ArchDoc] *E-Voting Architecture Document*, Version 1.1.0, Swiss Post Ltd., July 14, 2022
  ⇒ Updated to Version 1.2.0 on December 9, 2022 (see Appendix B)

For understanding the requirements defined in [OEV], [OEV Annex], and [ExpRep] as precisely as possible, we have mainly looked at the legally binding document versions in

[2] See https://gitlab.com/swisspost-evoting.

German. However, the terminology and citations used in this document are all taken from the available English translations.

Another important source of information for this report were the 2021 reports from other experts, particularly the ones from Bryan Ford, from Thomas Haines, Olivier Pereira, and Vanessa Teague, from David Basin, and from Aleksander Essex. For each of these reports, including our own reports on Scope 1 and Scope 2 [HKLD22b, HKLD22c], Swiss Post published individual response documents for discussing the findings that are not yet addressed in the current version. The references to the response documents of our own reports are the following:

- [ResScope1] *Response to examination report by BFH – Scope 1 Cryptographic Protocol*, Swiss Post Ltd., April 4, 2022.
- [ResScope2] *Response to examination report by BFH – Scope 2 Software*, Swiss Post Ltd., April 4, 2022.

In Section 2, we will look at the findings from our last year's assessment and discuss from our own perspective whether in the meantime they have been resolved adequately or not. The response documents published by Swiss Post were a useful starting point for this task.

1.2. Source Code

Since August 2021, Swiss Post publishes the complete source code of the system on gitlab.com. Corresponding repositories are located within the group swisspost-evoting, i.e., at the same location where the protocol and system documentations can be found.[3] Figure 1 shows a screenshot of the group's GitLab web page, which shows the system's three main components e-voting, crypto-primitives (including crypto-primitives-ts and crypto-primitives-domain), and verifier. Until recently, code updates for these components have been uploaded regularly, sometimes on a weekly or even daily basis. A first Version 0.15.0.0 of the two main cryptographic components crypto-primitives and e-voting have been released at the beginning of our assessment on June 18, 2022, whereas Version 1.0.0.0 of the verifier component is available since August 19, 2022. The overview of the GitLab commit histories in Figure 2 summarizes the updates released since then (note that not all releases were officially announced).

The frequent release of updates was a challenge during the first weeks of our mission, because it meant to re-examine areas of the code that we already analyzed. However, we tried to keep track of the relevant changes and adjusted our report whenever necessary. The statements made in Sections 1 to 3 of this report are therefore tied to the versions as listed in Table 1. The latest updates of the code from October, November, and December 2022 are discussed separately in Appendices A and B, respectively.

[3] See `https://gitlab.com/swisspost-evoting`.

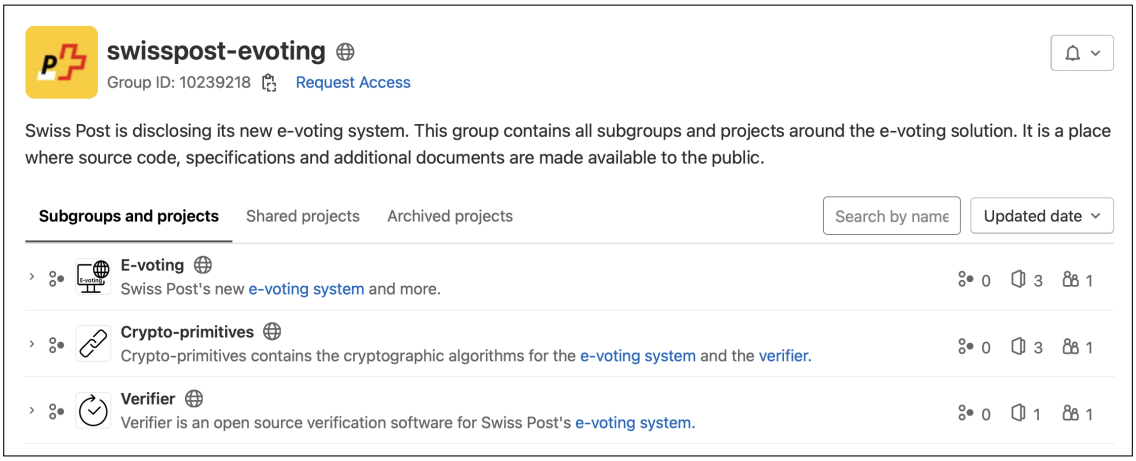


Figure 1: The GitLab project web page with links to the three main components.

Code Library	Version	Date	Commit (SHA-1 Fingerprint)
e-voting	0.15.3.0	July 26	`bd755b21d460bd3ca29b176c02d7d515e121afa5`
crypto-primitives	0.15.2.3	August 8	`510b31005abf1847e0e1add09d1c04441ea2fff7`
crypto-primitives-ts	0.15.2.3	July 20	`9c902e7cd372004fcd05bd6ef7705e864f38245a`
crypto-primitives-domain	0.15.2.5	August 12	`31283d5f57133fac36fa7f157c6f6a34517bb319`
verifier	1.0.0.0	August 19	`7aed568c5b98ed51b23738fad981316bac419291`

Table 1: Code versions of components as examined in this report.

The whole system code base is a huge collection of files. The core library e-voting, for example, consists of 2'068 Java and 293 JavaScript files with a total of more than 150'000 actual code lines. Table 2 gives an overview of the code base in terms of number of files and number of lines (code, comments, blanks). Compared to last years's code base, the total number of actual code lines has decreased by approximately 3.7% from 208'113 to 200'451. This is an indication that the refactoring process recommended in our last year's report has been conducted successfully, at least in certain areas of the code.

The e-voting component is a multi-module Maven project with a total of eleven Maven submodules. An overview of these submodules is given in Figure 3, which shows an excerpt of the component's pom.xml file. The components crypto-primitives, crypto-primitives-ts, and crypto-primitives-domain are (non-modular) Maven projects on their own, whereas the verifier component consists of four submodules. We were able to install and build them all in our IDE without considerable difficulties. During the building process, dependencies to external libraries were resolved automatically. The possibility of examining and running the code in our IDE without any broken dependencies was important for conducting our analysis efficiently.

The above screenshots, SHA-1 fingerprints, and code statistic represent the project state

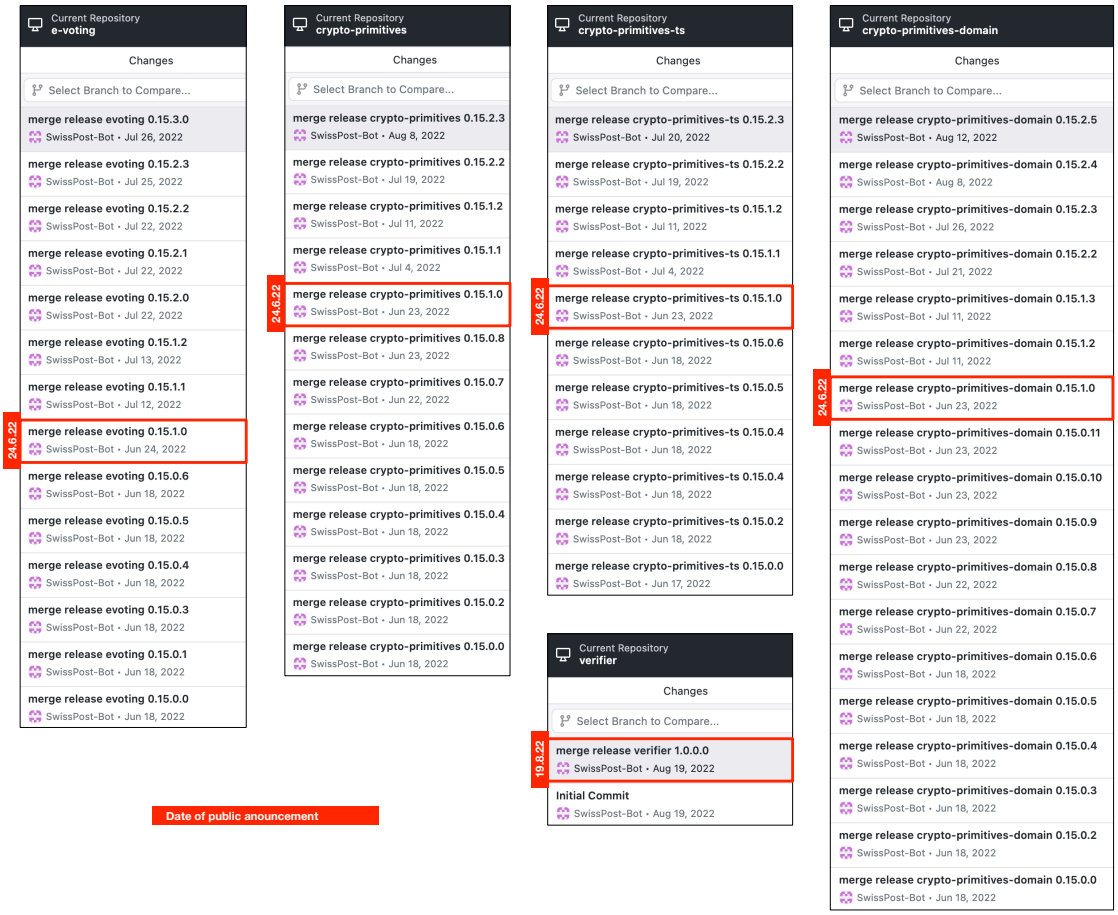


Figure 2: Commit histories of the GitLab projects with dates of public announcements.

in mid-August 2022. Our evaluation in Sections 2 and 3 refers to these versions. A major update of the whole project has been released on October 1, 2022, moving from pre-release versions 0.15.x.x to a first major release 1.0.0. This suggests that the project has reached an important milestone, indicating that the software has all major features and is considered reliable enough for general release. However, two other project updates have been released in early November and early December, moving from versions 1.0.0 to 1.1.0 and 1.2.0, respectively. At the time of finishing this report in January, 2023, we have already looked at these updates, but the discussion is contained in separate sections of this document (see Appendices A and B).

1.3. Purpose, Scope, and Overview of Examination

In a document called "*Audit Concept v1.4*", the Federal Chancellery describes the goals and rules for preparing, conducting and reporting the examination. This document, which is an updated version from last year's examination, has been given to both the

Code Library	Language	Files	Lines	Code	Comments	Blanks
e-voting	Java	2068	189966	125532	28912	35522
	JavaScript	293	38193	25978	5647	6568
crypto-primitives	Java	255	35392	22428	7578	5368
crypto-primitives-ts	TypeScript	90	19388	9284	7533	2571
crypto-primitives-domain	Java	98	9198	5701	1918	1579
verifier	Java	176	16862	10622	3702	2538
	JavaScript	3	379	325	34	20
	TypeScript	18	1010	581	330	99
	Total:	3001	310388	200451	55654	54265

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

```
<name>evoting</name>
<modules>
  <module>evoting-dependencies</module>
  <module>cryptolib</module>
  <module>domain</module>
  <module>command-messaging</module>
  <module>voting-server</module>
  <module>control-components</module>
  <module>secure-data-manager</module>
  <module>cryptolib-js</module>
  <module>voting-client-js</module>
  <module>voter-portal</module>
  <module>tools/config-cryptographic-parameters-tool</module>
</modules>
```

Figure 3: The file pom.xml from the multi-module Maven project `evoting`.

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 examination purposes and goals for all four

examination scopes. Relevant to our assessment are only the first two scopes:

> Scope 1: *"The protocol must fulfill the requirements listed in Chapter 2 of the annex of the [...] OEV."*

> 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."*

In a direct communication from the Federal Chancellery on April 13, 2022, the general goals of our mission were summarized as follows:

> *"We would like you to perform the examination and write an examination report on the following scopes:*
>
> 1. *Cryptographic protocol*
> 2. *Software:*
>
> *b) Assess the code quality and security*
> *c) Asses the documentation quality*
> *d) Assess the alignment between software development products*
> *e) Assess the implementation of the protocol*
> *f) Assess the functionalities"*

In our last year's mission, we put a strong focus on the cryptographic protocol (Scope 1), and so did many of the other experts. In Section 2, we take the documented findings from our report [HKLD22b] as a starting point for the re-assessment, with the goal of providing an updated overview of the protocol's critical parts and potential discrepancies to the requirements of the OEV and its annex.

In Scope 2, the main goal is to assess the code and documentation quality, the alignment between specification and code, and the correct implementation of the protocol. The results of our systematic and rigorous code analysis is documented in Section 3. We provide numerous recommendations and proposals for improvements to cope with the encountered problems. As already mentioned, in Appendices A and B we will revisit some of the raised issues in the light of the latest software and documentation releases.

1.4. Summary of Findings

In our first assessment of the Swiss Post Internet Voting System in 2021, we had the general impression of having examined an unfinished project. It seemed obvious that many critical parts of both the specification and the code were still under construction. The system implementation, for example, still included redundancies and inherited problems from earlier versions adopted from Scytl. The transition from these earlier versions to a clean, compact, and robust system and code base was clearly not yet completed.

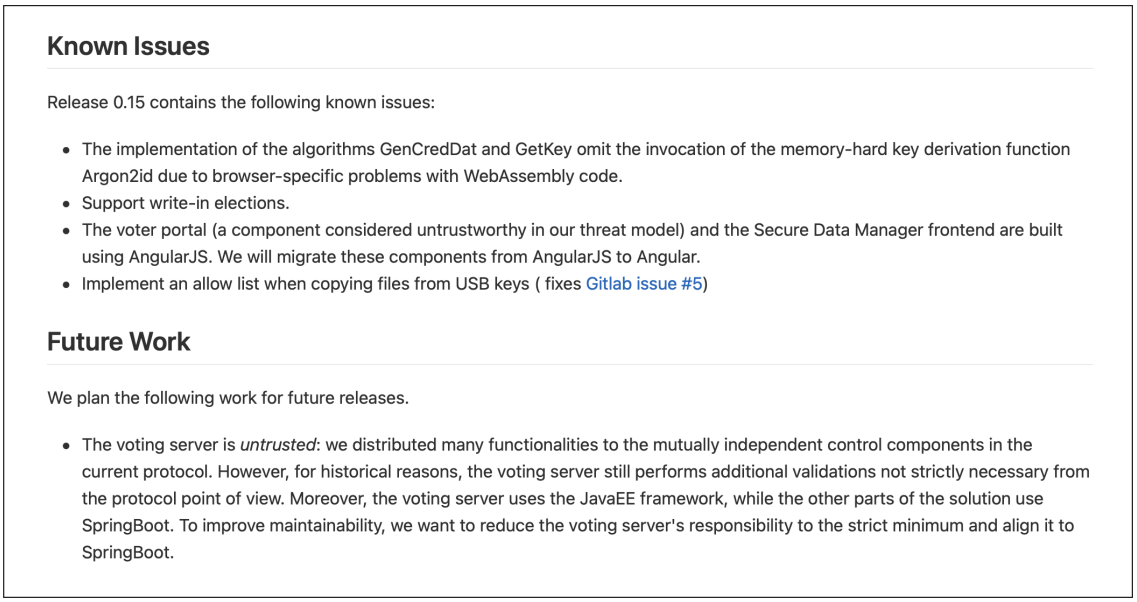
The new system and documentation versions that were available for the 2022 re-assessment are clearly a big step forward in this process. Many of the encountered problems have been either solved or substantially improved. For example, we were very pleased to see that the redundancies in the protocol specification, which resulted from spreading the protocol description and pseudocode algorithms across two different documents, have been eliminated. We were also glad to observe that the project structure and organization have been improved substantially in many parts, which now makes the code base much more accessible for inspections. It is still very large in size (approximately 200'000 lines of effective code), but for example locating the code implementing a specific pseudocode algorithm is no longer a noteworthy difficulty. Code readability has also benefited from removing complex frameworks such as Spring Batch from the cryptographically relevant part of the code. Furthermore, the whole Java code base has been migrated to Java 17.

These improvements, which result in large parts from the feedback of the 2021 examinations reports, demonstrate the importance of making everything publicly accessible and the benefit of involving international experts in the examination. This generates a fruitful platform for collecting feedback from many different external viewpoints. Relative to our last year's examination, we have slightly shifted the focus of our own viewpoint from conceptual aspects of the cryptographic protocol towards the details and quality of the code. Therefore, we spent great efforts in examining large parts of the code base on an almost line-by-line basis.

The main result of this thorough analysis is a relatively large list of mostly minor remarks with corresponding recommendations for improvements. We also detected some more fundamental problems in the software design, which from our perspective negatively impact the overall readability and quality of the code. Some of these design problems are related to conceptual questions, which possibly could be addressed more carefully. Furthermore, in a few areas of the specification and the code, we see a large potential for simplifications. Finally, we also encountered two components that—for different reasons—give the impression of being unfinished. In the following subsections, we shortly summarize our principal findings in each of the above categories.

Before starting the discussion of the principal findings, we must stress that we are fully aware of the Swiss Post's official statements about the next steps in their project roadmap, for instance in the main README.md file of the e-voting component, which contains a list of known issues and future work (see Figure 4). This list includes, for example, a

statement about the planned migration from AngularJS to Angular, which is exactly what we criticize in one of the topics discussed below. [updated in October release] Another official source for Swiss Post's project roadmap is the file Product_Roadmap.md from the documentation repository, which contains similar statements. Generally, we would appreciate if Swiss Post would commit to its roadmap for future releases even more firmly, for example by specifying expected release dates and feature lists.



Known Issues

Release 0.15 contains the following known issues:

- The implementation of the algorithms GenCredDat and GetKey omit the invocation of the memory-hard key derivation function Argon2id due to browser-specific problems with WebAssembly code.
- Support write-in elections.
- The voter portal (a component considered untrustworthy in our threat model) and the Secure Data Manager frontend are built using AngularJS. We will migrate these components from AngularJS to Angular.
- Implement an allow list when copying files from USB keys (fixes Gitlab issue #5)

Future Work

We plan the following work for future releases.

- The voting server is *untrusted*: we distributed many functionalities to the mutually independent control components in the current protocol. However, for historical reasons, the voting server still performs additional validations not strictly necessary from the protocol point of view. Moreover, the voting server uses the JavaEE framework, while the other parts of the solution use SpringBoot. To improve maintainability, we want to reduce the voting server's responsibility to the strict minimum and align it to SpringBoot.

Figure 4: List of known issues and future work from the project's main README.md file.

Source Code Analysis

In our systematic analysis of the source code, we put a special focus on detecting any sort of discrepancy between algorithm specification and source code. Given the fact that the alignment is now much more straightforward compared to last year's version of the system, we decided to look more at all the details by inspecting the code almost line by line. Unfortunately, we found that there are still numerous minor and mostly unnecessary discrepancies. Clearly, by applying a more careful attitude towards getting all the details right, most of them could have been avoided easily. All these discrepancies with recommendations for improvements are listed in the tables given in Subsections 3.2 to 3.4. Considered separately, a single entry from these tables is almost not noteworthy, but the sum of all the entries is what makes it a significant finding and something to improve in future versions.

Another problem detected during our code analysis is the lack of an adequate toolbox for dealing with mathematical objects such as tuples, vectors, and matrices. While they are fundamental for describing the protocol and the pseudocode algorithms in the specification documents, they are mostly implemented using standard Java data structures such as lists or maps. A problem closely related to this is the imperfect implementation of immutability, which is fundamental for making

code libraries robust, thread-safe, and easy to test. An example of this imperfect implementation is the widespread use of standard Java arrays for the representation and computation of byte sequences, instead of abstracting them into a proper immutable Java class. These are the topics, where we see a huge potential for improving the code quality at its very core. For a more detailed discussion, we refer to Subsection 3.2.1

Conceptual Problems

On a more conceptual level, we encountered a few problems that might be considered for further improvements. Something that affects both the specification and the code in the same manner is the lack of a proper definition of an algorithm's context. We agree with the general idea of distinguishing contextual parameters from actual parameters, but then the borderline between them should be made crystal clear. Unfortunately, this is not the case in the current version of the system. Some variables, for example, appear both as contextual and actual parameters, depending on the algorithm. Furthermore, it is not always clear how context variables are transmitted from one party to another and how their integrity is guaranteed. In the Java implementation, we have found many different ways of passing context variables to the methods implementing the algorithms. This inconsistent way of handling the context in the code, which seems to be the result of an unclear concept, considerably affects code readability. A detailed overview of the current context variables and a discussion of the existing problems can be found in Subsection 3.1.2.

Another important conceptual problem is the current election result consisting of decrypted (but not decoded) lists of prime numbers. This is clearly not the outcome that one would expect to obtain from a voting system. Here again, we think that a clear and more thoughtful definition of a fundamental concept is missing. This has some important consequences, for example when it comes to perform the verification process at the end of an election, which currently does not include the decoding. As discussed in Subsection 3.1.3, this may invite an attacker to modify or infiltrate the mandatory decoding table and thus completely circumvent the verification process.

Potential for Simplifications

Given the size of both the protocol specification and its implementation, the current system is undoubtedly a very complex construction. To improve the current situation, we were constantly looking for potential simplifications while conducting our analysis. Fortunately, we found quite a few of them in different areas of the specification and the code. Here is a quick summary of the most important simplifications:

- Set the Bayer-Groth shuffle proof parameters to $n = N$ and $m = 1$ and remove all unnecessary sub-algorithms from both the specification and the code.
- Remove the isProbablePrime pseudocode algorithm and all its sub-algorithms from the specification. [updated in November release: all algorithms removed]
- Eliminate the voting server from the protocol specification by delegating its

two simple tasks to the voting client.

- Substitute the terms return-code control component (CCR) and mixing control-component (CCM) by *control component* (CC).
- Eliminate the primes mapping table pTable from the specification and the code. Instead, call a deterministic algorithm at each occurrence, which derives the primes from the group parameters.
- Remove the unnecessary encryption layer around the partial choice return codes, which is still present "*for historical reasons*" [SysSpec, Section 5.1.4].
- Either implement write-ins comprehensively, or remove them completely. In the latter case, replace the implementation of multi-recipient ElGamal encryption by ordinary ElGamal encryption.
- Use the ballot box IDs bb uniquely for context separation purposes in the zero-knowledge proofs. Eliminate corresponding lists $\mathsf{L}_{\mathtt{bb},j}$ and $\mathsf{L}_{\mathtt{bb},\mathtt{Tally}}$ from the protocol (currently they create unnecessary side-effects in some algorithms).
- Migrate the remaining legacy code from the cryptolib and cryptolib-js libraries, following the statements "*This library is deprecated [...]*" and "*no longer used for the implementation of the protocol*" [ArchDoc, Section 3.2 and 5.10.1].

Note that most of the above simplifications can be implemented without losing any of the necessary functionalities or properties of the system (some of them are dispensable relics from the earlier Scytl system). Clearly, implementing them all in a strict and comprehensive manner will greatly decrease the complexity and size of the current system. This would help to further improve the auditability of both the protocol specification and the source code, and it would follow the "*Keep things as simple as possible*" (KISS) principle as imposed in [ArchDoc, Section 8.1].

Unfinished Components

The current JavaScript implementations of the voter-portal and secure-data-manager modules still depend on the outdated AngularJS framework, which has officially reached the status of an end-of-life (EOL) product on December 31, 2021. Besides not getting further updates or support in the future, there are still known vulnerabilities in Version 1.8.2 from October 21, 2020 (currently used in the secure-data-manager implementation), and in the final Version 1.8.3 from April 8, 2022 (currently used in the voter-portal implementation).[4]

As the following statements from the architectural document, the README.md file from the e-voting repository, and the response document to our last year's findings show, this is a known but still open issue:

> "*Migrate the voter portal and secure data manager from AngularJS to Angular as AngularJS is end of life as of January 2022.*" [ArchDoc, Section 9.1.6]

[4] See `https://security.snyk.io/package/npm/angular`.

> "*The voter portal [...] and the Secure Data Manager frontend are built using AngularJS. We will migrate these components from AngularJS to Angular.*" [README.md, Known Issues]

> "*The migration will happen before the e-voting system goes live, but it has not been addressed so far for priority reasons.*" [ResScope2, Section 2]

The fact that the migration from AngularJS to Angular is still pending in the system under evaluation is quite surprising. [updated in October release] We think that the voting client, because it is trustworthy for vote privacy by definition, should get special attention in any security-related respect, including the management of its dependencies. We agree that the dependency to AngularJS is not directly related to cryptography, but using a EOL framework for building security-critical software in domains such as e-voting is clearly against all rules of best practice. It furthermore violates one of the project's general architectural principles: "*Maintain dependency freshness and regularly advance to the latest version of dependencies*" [ArchDoc, Section 8.1].

The second apparently unfinished component is the verifier, even if the current version number 1.0.0.0 suggests something else. For example, we were quite surprised to receive an update of the verifier specification on August 19, i.e., in the middle of our assessment period, which included substantial changes such as an additional pseudocode algorithm. The latest version of the verifier specification, however, is still underspecified, with certain pseudocode algorithms containing vague textual descriptions of certain tasks instead of precise executable instructions and operations. According to the system's software development process as illustrated in [ArchDoc, Figure 28], the definition and improvement of the algorithm specification always comes in the first place, not the reviewing and adjustment of the code. In case of the verifier, where the specification lags behind the code at different places, we observed the inversion of this fundamental development principle.

2. Review of Previous Findings

In this section, we review some of the principal findings from our last year's reports. As already mentioned, many of the encountered problems have been adequately addressed in the current system. Put together, these improvements provide a big step towards a system that eventually may be approved to fulfill the OEV requirements. While we acknowledge the implemented improvements and appreciate the general direction the project has taken, we regret that not all of our concerns have been taken into account. In the response documents for Scope 1 [ResScope1] and Scope 2 [ResScope2], Swiss Post addresses the unresolved findings and explains whether and how they will be taken into account in the future according to their analysis. We understand that findings of lower priority can possibly be postponed, but we also think that there are still some findings of higher priority, which should have been addressed in the current system.

In this section, we follow the given structure of our previous reports to re-examine the findings one by one in the light of the current system and the responses received from Swiss Post. In Subsection 2.1, we walk through the findings related to Scope 1 (Cryptographic Protocol) to see whether and to which degree they have been addressed, and we do the same in Subsection 2.2 for the findings related to Scope 2 (Software). In both cases, we will put the focus on the most critical open problems.

2.1. Scope 1: Cryptographic Protocol

In our 2021 report on Scope 1, we structured the results of our analysis into six areas of "Critical Findings" [HKLD22b]. While we observe major improvements in three of them, there are still at least two areas in which our concerns have not been addressed at all. Under "*Proving Vote Abstention*", the origin of the problem comes from the definition of the corresponding OEV requirement, which arguably leaves some room for interpretation (see Subsection 2.1.3), and this may justify the Swiss Post's decision not to address it. Under "*Vote Privacy of Voters With Restricted Eligibility*", however, we think that there is no room for interpretation that would justify the current sub-optimal solution. From our current perspective on the system after conducting the re-assessment, this is the main open problem in Scope 1 from the findings listed in our report.

2.1.1. Missing Update to Draft OEV

At the time of the examination in 2021, the protocol specification was not yet updated to the draft of the new OEV, which was already available during the assessment period. This lack of synchronization created an awkward situation for the reviewers, because it meant to evaluate the alignment of two documents, which were not directly meant to be aligned. Therefore, we were forced to evaluate the alignment on a best-effort basis based on our

pre-knowledge about the protocol and its most recent adjustments. Retrospectively, this was a major challenge in conducting the 2021 assessment.

By eliminating all the quotes and references to the old OEV and by adjusting the terminology accordingly, this problem has been solved completely in [SysSpec, Version 0.9.7] and [ProtSpec, Version 0.9.11] from October 15, 2021. Therefore, we fully agree with the response given in [ResScope1, Section 2]:

> *"First, the BFH determined that the protocol contains references to an obsolete version of OEV. In version 0.9.11 of the cryptographic protocol, published in October 2021, Swiss Post updated all references in line with the latest draft of OEV. This finding by the BFH has therefore been resolved and implemented."* [ResScope1, Section 2]

2.1.2. Role of Auditors

According to the updated OEV, the auditors are explicitly allowed to perform their checks "*during the setup phase*" and "*after tallying*", but not in any of the other protocol phases. In the protocol version from last year, however, the auditors were also involved in the tally phase. This violation of the permitted responsibility was our main objection against the suggested role of the auditors. In the current version of the protocol, by re-organizing the process in the tally phase, this problem has been solved.

Other problems that we encountered were about the auditors' decision making as a group and the reporting of a detected failure. These problems are partially solved by excluding the auditors from the tally phase, but they still exist at the end of the setup phase. Figure 5 shows an excerpt from [SysSpec, Figure 6], which defines the auditors' task of verifying the configuration data received from the setup component. Interestingly, the result of this verification is *only* communicated to the setup component itself. This raises the following two questions:

- To the best of our understanding, the setup component is supposed to stop the process in case of a reported failure, but what if the auditors' response is simply ignored?
- According to [OEV Annex, Section 2.2], the communication channel between the setup component and the auditors is unidirectional, so how can the auditors' response be communicated to the setup component as depicted in Figure 5?[5]

Generally, we believe that here the protocol description is still not sufficiently accurate with respect to handling the case of a failed verification. This includes the case of a disagreement between the auditors, which must be resolved in one or the other way,

[5] We are aware of [ExpRep, Ziff. 2.1], where the auditors are allowed to carry out specific verification tasks for the setup component. However, it remains unclear how to proceed with negative outcomes of the tasks, as there is no back channel.

presumably by an offline dispute resolution procedure conducted by humans. We are aware of the response given in [ResScope1, Section 4.1] and the discussion of Issue 13 in the documentation GitLab repository, and we agree that disputes between auditors can be resolved unambiguously, but we recommend addressing this important topic in the system specification to clarify all the questions raised above.

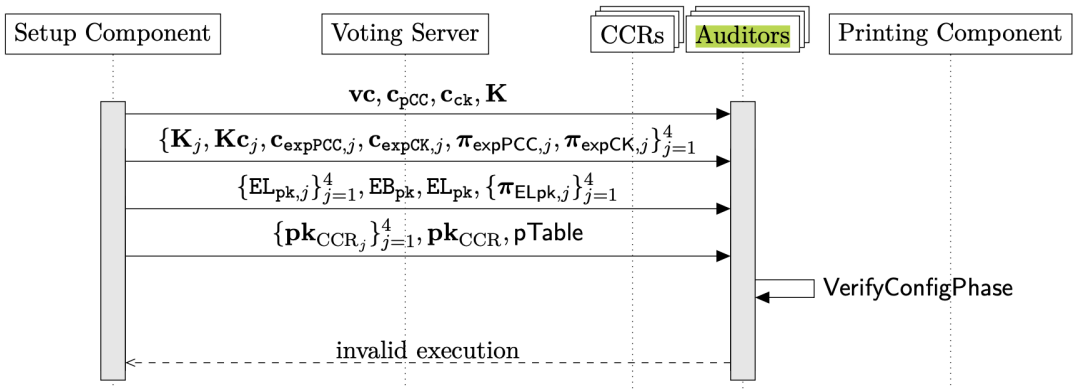


Figure 5: The role of the auditors at the end of the setup phase according to [SysSpec, Figure 6].

2.1.3. Proving Vote Abstention

Our proposal to introduce vote abstention codes, based on a strict interpretation of the OEV requirement that "*a voter who has not cast his or her vote electronically can request proof [. . .]*" [OEV, Art. 5.2c] that no vote has been cast on behalf of the voter, has not been taken into account. The main reason is the following statement from the Federal Chancellery's explanatory report:

> *"For reasons of efficiency, it is sufficient for the competent cantonal office to confirm to the voter that no vote has been cast on their behalf."* [ExpRep, Section 4.2.2]

We agree that this statement leaves enough room for interpretation to dismiss our proposal. The following explanation for the decision not to consider abstention codes was included in a draft of [ResScope1] from March 2, 2022. We regret that this statement has been deleted from the final version, because it made the legitimate reasons for dismissing our proposal transparent (with "*highlighted statement above*", it refers to the above statement from the explanatory report):

> *"We will not implement abstention codes like the BFH proposed for the time being. We base our protocol and solution on the highlighted statement above."*

Even if we do not question the legitimacy of this decision, we still believe that implementing the confirmation of the "competent cantonal office" could be a big challenge in practice, not only for the cantons operating this office, but also for the Swiss Post who will need to equip them with the necessary tools. Unfortunately, no information on this is given in any of the publicly available documents. All things considered, we still think that implementing abstention codes would be the better and more general solution.

2.1.4. Underspecified Protocol Aspects

A major general issue in the specification documents from last year was the lack of sufficient technical preciseness in some areas of the protocol, which left too much room for interpretation. In the new version, great progress has been made in this matter, partly by improving the explanations given in the documents, by simplifying certain technical components, or by eliminating relics from earlier versions. In the remaining of this subsection, we quickly review each of the topics raised in our report from last year.

Primes Mapping Table

The so-called *primes mapping table* pTable was completely unspecified. It appeared magically at various places of the protocol. In [SysSpec, Section 3.4.2], a formal definition of pTable as an injective mapping of voting options to prime numbers from $\mathbb{G}_q$ is now given, and there are two pseudocode algorithms EncodeVotingOptions and DecodeVotingOptions specifying the encoding and decoding of voting options based on pTable. Unfortunately, these algorithms are never explicitly called in the protocol, and we also couldn't find their implementations in any of the system components. This topic is therefore an example of a problem that has been addressed, but not to a satisfactory degree. We will further discuss this topic in Subsection 3.1.3, where we argue that pTable could be completely eliminated from the protocol without loosing anything. This proposal is also included in the list of recommended simplifications from Subsection 1.4.

Electoral Board Key Pair

The role of the electoral board in the protocol as a group of people receiving a private key from the print office was problematical in various respects. With the introduction of the *setup component* and the *tally control component* in [SysSpec, Section 2.8], the solution is now quite different. Based on the human-generated passwords received from the members of the electoral board, the setup component generates the public key $\mathtt{EB}_{\mathtt{pk}}$ by calling SetupTallyEB (which internally generates corresponding shares of the private key), whereas the tally control component derives the private key $\mathtt{EB}_{\mathtt{sk}}$ by calling MixDecOffline. The corresponding information flow is depicted in [SysSpec, Fig. 7] and [SysSpec, Fig. 7], respectively. Assuming a trustworthy setup component, who has no intention to exploit its knowledge of the private key, this is an acceptable solution. Note that the introduction of an offline tally component is not an explicit OEV requirement, but Swiss Post defines its purpose as follows:

"*We leverage the electoral board as an additional operational safeguard enforcing the four-eyes principle for the final decryption of the votes*" [SysSpec, Section 2.5].

Ballot Box

In earlier versions of the protocol specification, the term *ballot box* was used very inconsistently. A clear definition seemed to be missing. In the new specification document, a ballot box is an abstract notion for two lists, a list of *unconfirmed votes* and a list of $\mathtt{N_C}$ *confirmed votes* (see definition of election event context in [SysSpec, Figure 5]). Note that whole electorate is divided into $\mathtt{N_{bb}}$ sub-groups (according to given electoral constituencies), and the protocol is executed for each sub-group separately. Each protocol execution defines its own ballot box, which are identified over their *ballot box IDs* $\mathtt{bb}_j$ (Base16 strings of length $\mathtt{l_{ID}}$). This means that only one ballot box ID, usually denoted by `bb`, is relevant for a given protocol run. Using this identifier for context separation purposes in zero-knowledge proofs (together with the *election event identifier* `ee`) corresponds to a best practice. So far, our objection from last year's report has been addressed properly.

A remaining problem comes from the lists of $\mathsf{L}_{\mathtt{bb},j}$ and $\mathsf{L}_{\mathtt{bb},\mathtt{Tally}}$, which contain the IDs of the shuffled and decrypted ballot boxes. These lists are kept and updated by the mixing and the tally control components, respectively. Currently, the updating takes place as a side-effect of calling the algorithms MixDecOnline and MixDecOffline. This is clearly a remaining conceptual mistake, because these lists are completely irrelevant when each control component verifies the shuffle and decryption results of the preceding control parties. Removing them is one of our recommended simplifications in Subsection 1.4. Furthermore, we think that "ballot box ID" is somewhat misleading, because each authority keeps its own ballot box, and they are not always identical at all times, so `bb` can not be an identifier for all of them. We suggest to replace it by "electorate ID" or something similar (the term "ballot box" would then disappear almost entirely).

Logs

A relic from the original Scytl system was the concept of a *secure log*. In the version from last year, these logs were largely unspecified, but they were used everywhere to represent the parties' state of knowledge at different stages of a protocol. In the current version of both the protocol and the code, this concept and the term "log" have been removed entirely.

Keystore and Start Voting Key

The protocol from last year did not specify when and how the voting client receives the keystore $\mathtt{VCks_{id}}$. Now it is clear that the setup component uses GenCredDat to generate the list $\mathbf{VCks}$ of all keystores and then sends them to the voting server [SysSpec, Figure 6]. At the beginning of the voting session, i.e., after successfully authenticating to the voting server, the keystore is sent to the voting client. We regret that these initial messages are not shown in [SysSpec, Figure 8].

Another objection was related to the generation and the parameters of the *start voting key*. The additional information given in [SysSpec, Section 3.5] shows

that the awareness for this important issue is now given, but the statement below Table 11 about fixing the Argon2 parameters to "*reasonably low values*" is not a very good response to our objection, especially since a factor of 1000 only corresponds to approximately 12 bits, instead of the required 16 bits.

Authentication and Context Separation

Authentication was not sufficiently well described in earlier versions of the protocol specification. By providing a comprehensive list of all the signed protocol messages in [SysSpec, Tables 15–17], this problem has been properly addressed. The given overview shows that now the setup component, the online control components, and the tally component systematically sign their outgoing messages, and it defines for each signature some *context data*, which is important for solving the context separation problem. Another solved problem is the size of the RSA signature keys, which has been increased to 3072 bits for all security levels.

With respect to authentication and message integrity, we found two remaining problems in the current protocol description. Something that still remains unclear is the authenticity and integrity of the election event context. The problem here seems to be more fundamental, because it is generally unclear how the context is defined and transmitted to all participating parties. We will further discuss this problem in Subsection 3.1.2, but we already want to stress that this is something that could undermine the security of the system very profoundly.

The second remaining problem is the fact that the untrusted voting server also signs two types of messages (called VotingServerEncryptedVote and VotingServerConfirm, see [SysSpec, Table 16]). We do not see the benefit of these signatures, because receiving a signed message from an untrusted voting server has the same quality as receiving a message from an unknown adversary. In Subsection 3.1.4, after discussing this issue more profoundly, we will recommend removing the voting server from the protocol description.

Election Use Cases

A description of the supported election use cases and a clear definition of the resulting election parameters was missing in earlier versions of the available specification documents. In [SysSpec, Section 3.3 and 3.4.1], the election parameters are now relatively clear. The main two parameters are the number of voting options n and the number of selectable voting options ψ satisfying $1 \leqslant \psi < n$ (the third parameter $\hat{\delta}$ is irrelevant as long as write-in options are not supported). Therefore, without giving further information, some readers may still conclude that the protocol only supports single ψ-out-of-n elections, even if this is not at all the intention of the system designers.

What is missing in the discussion is the fact that elections are often held simultaneously and that in simultaneous elections only a subset of the $\binom{n}{\psi}$ combinations of voting options are allowed. In case two referendums are held in parallel, for example, a combination that assigns two options to the first and zero options to the second question is clearly an invalid vote. Unfortunately, the mechanisms for exclud-

ing such invalid combinations of voting option is discussed separately in [SysSpec, Section 3.4.3]. We still think that these two discussions should be merged into one, because otherwise it seems difficult to grasp the whole complexity of this topic. Finally, for further clarifying the discussion of this topic, we recommend replacing the elements of the current election event model (`correctnessID`, **ciSelections**, **ciVotingOptions**) and the two Algorithms 3.4 and 3.5 by a more mathematical approach, like the one that we outlined in our last year's report.

In the current protocol documents, a description of the supported election use cases from the Swiss context is still missing. A reference to a file ElectoralModel.md on gitlab.com, which lists five different types of supported elections, is given in [ResScope1, Section 4.2], but it does not describe the mapping of arbitrary combination of such election types into the above election event model. The same file also mentions the existence of "*a set of Excel sheets containing all the features of an election*", but a reference to these files is not given. Finally, [ResScope1, Section 4.2] contains a promise that has not yet been delivered: "*However, we will enhance the protocol documentation in a future version to address the BFH's finding*".

Write-Ins

In this question, it seems that we have reached some sort of a deadlock between the Swiss Post, the Federal Chancellery, and us. In our report, we have expressed our concerns that the inclusion of write-ins may have an impact on accepting a submitted ballot as valid or not, for example in case of a malicious voter submitting a vote for regular candidates in combination with non-empty fields for write-in candidates. In [ResScope1, Section 4.3], a reference to a statement from [ExpRep, Section 4.2.1] is given as a justification for not addressing this problem, but the quoted statement only refers to the "*blank text fields*" for entering the write-in candidates, not to the ballot as a whole. According to a direct communication, the Federal Chancellery seems not to be worried much about our particular attack scenario, but this standpoint does not match with our understanding of the OEV and its explanatory report. An official clarification in either direction could contribute to finding a solution for the current deadlock.

Another unsatisfactory situation is the current half-hearted implementation of write-ins in both the protocol and the code. On one side, using multi-recipient ElGamal encryptions, the ballots are ready to include votes for write-in candidates, and so is the processing of these ballots during the tally phase. But on the other side, votes for write-in candidates are completely ignored at the end of the election process (see Subsection 3.1.3). Given that Swiss Post responds in [ResScope1, Section 4.3] with "*We do not plan to remove the write-ins, as this would exclude too many elections in Switzerland, [...]*" to our recommendation to exclude write-ins from their current system, we would have expected them to finish the implementation and doing it properly. Generally, we think that unfinished or inactive functionalities should not be the subject of any examination.

2.1.5. Vote Privacy of Voters With Restricted Eligibility

In case of a heterogeneous electorate, we learned from [SysSpec, Section 3.3] that voters with different voting rights are grouped into separate subsets (called *verification card sets*), such that in each subset, all voters have the same voting right. The protocol, which by design cannot handle different voting rights in a single protocol run, is then executed separately for each subset. This is what we have criticized in our report, because introducing such subsets may considerably diminish corresponding anonymity sets. In an extreme case, the election result computed separately may then reveal sufficient information for completely breaking the secrecy of all submitted votes.

The example given in [ResScope1, Section 4.4], where an electorate consisting of 1030 voters is divided into three groups of 1000, 20, and 10 voters, illustrates this problem. In this case, election results are computed separately for each group using three simultaneous protocol executions. In the smallest group of 10 voters, the size of the anonymity set has been reduced by a factor of 103 compared to the original anonymity set. Vote privacy is therefore violated accordingly, but only due to technical reasons.

The above example also shows the awareness of Swiss Post for this problem. However, given the following statement from the response document, we think that there is still some misconception with regard to the notion of privacy in electronic voting protocols:

> *"However, the described attack does not violate standard definitions of voting privacy [...]: there is always the risk of a privacy breach if a ballot box contains too few votes."* [ResScope1, Section 2]

The above quote also includes a reference to one of the most cited papers on formal definitions for privacy in secure voting protocols [BCG+15]. Here is a quote from that paper, which exhibits the misconception in the above statement:

> *"Recall that ballot privacy attempts to capture the idea that during its execution a secure protocol does not reveal information about the votes cast, beyond what is unavoidably leaked (e.g. what the result of the election leaks)."* [BCG+15, Chapter IV]

The point here is that a privacy breach can only occur, if the set of eligible voters has been diminished artificially, because this automatically leaks information beyond what is unavoidably leaked. But this is exactly what is happening in the current Swiss Post voting system in examples like the one discussed above. In the light of our proposal for a better solution in [HKLD22b, Section 2.5], we think that any sub-optimal solution that is exposed to this problem cannot be justified.

In their response to this finding, Swiss Post argues that the cantons usually group the Swiss citizens living abroad into one sufficiently large cantonal counting circle. But some

cantons may have other exceptions, for example young voters under 18 or permanent foreign residents. In such cases, it may not always be possible to guarantee a minimal size for the artificial counting circles, especially since a new voting channel will always require time to gain popularity. Therefore, we want to stress again the importance of this topic and recommend that Swiss Post will not wait too long for keeping its promise to "*implement it as proposed by the BFH in a later phase*" [ResScope1, Section 4.4].

2.1.6. Legitimacy of Proof

Given the restricted time for conducting this assessment, we were not able to review the security analysis in [ProtSpec, Version 1.0.0]. The changes made to the document are listed in the file README.md on the documentation GitLab repository. They suggest that several general improvements and clarifications have been made to address the feedback received from the reviewers. The most obvious improvement is the removal of the redundant protocol description in Section 10, which has been moved entirely to [SysSpec] and [CryptPrim]. This simplification is very useful, because now it is clear that the proof always refers to the real protocol. Our main general comment from last year's report, namely that in some aspects the proof was too strongly abstracted from the real protocol, is therefore no longer valid. Apart from that, we cannot make any statements about the status of the updated proof document.

2.2. Scope 2: Software

As the focus of our current mission has been on analyzing the code of the implemented software, we can keep this subsection on reviewing the Scope 2 findings from last year shorter than the previous section on Scope 1. The results of our analysis in Section 3 give a detailed overview of the current situation and list all the encountered problems. Compared to the version from last year, the overall alignment between code and specification has been augmented considerably. There are also many apparent improvements of the code quality, which makes the complex code base more accessible for auditors.

In Subsection 1.4, we have already summarized the remaining shortcomings of the current system implementation. One general observation is the existence of a large number of mostly minor discrepancies between pseudocode and Java code, which could have been avoided by placing greater emphasis on the details. Almost all parts of the code are affected by this. Another fundamental problem is the limited functionality of the current mathematical toolbox, which is often an obstacle for achieving a better code quality. From the other findings in our last year's report, the synchronization problem has been solved, whereas the potential vulnerability in case of a poor entropy source still exists.

In the review of the previous findings given below, we follow the structure and topics of our last year's report [HKLD22c, Section 2].

2.2.1. Deviations Between Protocol and System Specification

The system documentation from last year contained two descriptions of the cryptographic protocol, one with less details (descriptive algorithms) and one with more details (pseudocode algorithms). In our report, we listed numerous discrepancies between the two documents, for example with respect to the protocol's message flow or the computations performed by the algorithms [HKLD22c, Section 2.1]. By completely deleting the high-level protocol and algorithm descriptions from [ProtSpec], this problem has been solved. The resulting removal of redundancy makes the current system documentation much more accessible.

2.2.2. Deviations Between System Specification and Source Code

The code inspected during our last year's mission seemed to be in a transitional state. Certain parts of the code had obviously undergone a major revision compared to earlier versions. Those parts were already very well aligned with the specification, whereas in other parts, this alignment had not yet been realized. In the new version, it seems that the transition process has been completed in almost all parts of the system, which means for example that names of classes, methods, parameters, and local variables can now be linked easily to the specification, and that there is a close match between the executed code and pseudocode statements. In their response to our report, Swiss Post promised to prioritize the alignment between code and specification:

> *"The alignment work is at the top of our priorities and will be achieved gradually, starting with the central trustworthy components."* [ResScope2, Section 2]

While we acknowledge the vast progress made in this regard, there are still many small and mostly insignificant differences between code and specification, which prevents the matching from being perfect (see Subsections 3.2 to 3.4).

Another problem in the previous version of the system was the existence of source code that seemed vital for the implementation of the cryptographic protocol, but for which no pseudocode existed. The cleansing procedure was the most significant example for this type of discrepancy, but this particular element of the implemented protocol does not exist anymore. In the current implementation, we found a few places where the verifier specification still lags behind the code (see Subsection 3.4), but otherwise this problem seems to be solved (see Subsection 2.2.4).

2.2.3. Crypto-Primitives

In our analysis of the former crypto-primitives component, we made a few remarks on different aspects. Many of them have been addressed in the current version:

- Preconditions are now consistently listed under the heading **Require**. In most cases, they are implemented in a strict and systematic manner.
- In the implementation of the proof generation and verification algorithms, the concatenation from $\mathbf{i}_{\mathsf{aux}}$ to $\mathbf{h}_{\mathsf{aux}}$ has been replaced by composition.
- The interface `CryptoPrimitives` and the class `CryptoPrimitivesService` have been removed.
- Some inconsistencies in the method names have been eliminated.

We also made some recommendations to simplify the code of the component, for example by reducing the Bayer-Groth proof to the special case $m = 1$ and $n = N$, by introducing an explicit context object that is passed to all methods as additional parameter, or by defining the algorithms as static methods which operate on pure data objects. These recommendations have not been implemented. In Subsection 1.4, some of them appear again on the current list of recommended simplifications.

2.2.4. Underspecified Concepts and System Components

In our report from last year, we also made a few remarks about some components that were apparently relevant for the cryptographic protocol and its implementation, but which were not sufficiently well specified. Some of these components have been simplified or removed:

- Objects of type `SecureLog` were included in the implementations of the CCR and verifier components, but the concept of a *secure log* was not well thought out. In the meantime, secure logs have been removed from both the protocol specification and the code (see Subsection 2.1.4).
- The sharing of the electoral board's private key $\mathtt{EB}_{\mathtt{sk}}$ using algorithm SplitSecretShares was very unclear. In the current protocol, $\mathtt{EB}_{\mathtt{sk}}$ is derived from a set of human-generated passwords, which makes the sharing procedure and algorithm obsolete.
- Another unclear procedure was the transmission of the keystores and other election data to the voting client at the beginning of the voting process. As already discussed in Subsection 2.1.4, the given explanations are now much better, even if the exact data flow is still not shown in the protocol diagram of the voting phase [SysSpec, Figure 8].
- Symmetric encryption was not sufficiently well specified. In the pseudocode algorithms GenCiphertextSymmetric and GetMessageSymmetric, too many details remained unspecified. This problem has been solved in the new versions of these algorithms.

We also remarked that the exact shape and the initialization of the data structures CMTable, pTable, and **correctnessID** were not always very clear. Unfortunately, this is still the case in the current version of the system. For example, a definition for CMTable is still not present in the whole specification document, whereas [SysSpec, Section 3.4.4] about the **correctnessID** has not been improved much.

In the same comment, we also mentioned a lack of clarity with respect to the different lists, which are managed by the control components for keeping track of the processed votes. Unfortunately, they are still defined at different places of the document, and in each case only a single sentence is given as explanation. Here is an overview of these definitions (note that an additional list $\mathsf{L}_{\texttt{genVC},j}$ and a key-value map $\mathsf{L}_{\texttt{confirmationAttempts},j}$ are present in the new version of the protocol):

[SysSpec, Section 4.1.3]: Algorithm GenVerDat

- "*Partial choice return codes allow list*" $\mathsf{L}_{\texttt{pCC}}$

[SysSpec, Section 4.1.4]: Algorithm GenEncLongCodeShares

- List of voting cards for which the control components decrypted the partial choice return codes: $\mathsf{L}_{\texttt{decPCC},j}$
- List of voting cards for which the control components already generated long choice return code shares: $\mathsf{L}_{\texttt{sentVotes},j}$
- List of confirmed votes: $\mathsf{L}_{\texttt{confirmedVotes},j}$
- List of processed voting cards: $\mathsf{L}_{\texttt{genVC},j}$
- Key-value map of number of confirmation attempts per voting card: $\mathsf{L}_{\texttt{confirmationAttempts},j}$

[SysSpec, Section 4.2.2]: Algorithm SetupTallyEB

- List of shuffled and decrypted ballot boxes: $\mathsf{L}_{\texttt{BB},j}$

With respect to the implementation of these data structures, a more profound discussion of the implemented general concept (keeping track of the current state) and an overview like the one given above, possibly with better explanations for each item in the list, would greatly improve the clarity of both the specification and the code.

2.2.5. Quality of Code

Measuring the quality of the code is generally a difficult task. On one side, there are standardized measures for evaluating certain quality aspects, and great tools exist for deriving these measures automatically from the code base, but on the other side, obtaining good scores from these tools is often not the most important criterion. Knowing that others had already analyzed the Swiss Post system using such tools, we decided not to repeat this task. Instead we discussed a few problematical topics that we encountered while looking at the code. In their response to our report, Swiss Post has underlined their commitment to attain the best possible code quality:

> *"Swiss Post is working intensively to continuously improve the system. Any software project has to be maintained to ensure the best quality and security."* [ResScope2, Section 2]

From the topics discussed in our report, some have been addressed in the current version and some not:

- The validation of input parameters was implemented very inconsistently, but this is no longer the case in the current system. The only remaining problem is the lack of a clear definition and consistent implementation of the context variables (see Subsection 3.1.2).
- The existence of two very similar cryptographic libraries was very confusing. Given that most functionalities have now been moved permanently from cryptolib to crypto-primitive, this problem is almost entirely solved. However, we regret that some remaining legacy code from cryptolib is still present (see recommended simplifications in Subsection 1.4).
- In some cases, unusually long names (up to 45 characters) were used for classes, methods, and variables. This affected the readability of the code considerably. We still encountered some relatively long names in the current code base, but the most extreme cases have been eliminated.
- Another obstacle for better code readability was the wide-spread application of the Spring Batch framework and the large number of external Java and JavaScript libraries. By removing Spring Batch from the cryptographically relevant part of the code, the first problem is solved. However, the number of external dependencies is still very large, for both Java (see pom.xml in the module evoting-dependencies) and JavaScript (see for example package.json in voter-portal).
- Our last comment was about the using Java 1.8 and AngularJS, which were both close to reaching their end-of-life status. By migrating the code to Java 17, the first problem is solved, but the announced migration to Angular is still pending (see Subsection 1.4). [updated in October release]

> *"Only one single dependency remains outdated (AngularJS), and it is important to mention that this dependency is not related to cryptography. The migration will happen before the e-voting system goes live, but it has not been addressed so far for priority reasons."* [ResScope2, Section 2]

In our current mission, by re-examining almost all parts of the code very carefully, we encountered a few other code quality problems. The most obvious one is the sub-optimal mathematical toolbox, which does not offer proper abstractions for important mathematical objects such as tuples, vectors, or matrices (see Subsection 3.2.1). The poor implementation of immutability is a related open problem.

2.2.6. Synchronization

In our 2021 report on Scope 2, we explained in detail why the synchronization of messages and their processing in a distributed system is fundamental for the correctness and robustness of the overall system [HKLD22c, Section 2.6]. During the voting phase, for example, a malicious voter or voting client may try to cast multiple ballots at the same time, either to learn the choice return codes for more than one ballot or to provoke an inconsistent state on the server. In the response to our report, Swiss Post mentioned that they had already been working on this and that they wanted to further improve the implementation and documentation:

> *"In release 0.13 (February 2022), we implemented a mechanism at database level to ensure that the control components process messages exactly once, thus preventing these types of synchronization attacks. We will further improve this mechanism and document it properly for the next audit."* [ResScope2, Section 4.2]

In the current version of the available documents, we found a discussion of this topic and a description of the implemented solution in [ArchDoc, Section 10.1.2]. From analyzing the proposed synchronization mechanism and its implementation, we conclude that this problem has been solved adequately.

To ensure high availability of the system, multiple instances of each control component run in parallel. This makes synchronization even more difficult, because it involves multiple concurrent instances of the same component. This implies that the current state of a component must be managed across all instances, for example by using a shared database. The proposed solution of delegating the synchronization problem to the database is therefore straightforward. In their implementation, Swiss Post uses classical techniques such as optimistic and pessimistic locking. They also implemented a property called *exactly once processing*, which guarantees that external messages are processed exactly once. This prevents not only identical messages from being processed more than once, but also similar messages (of the same type). For example, if a voter sends different ballots simultaneously, then these ballots are recognized as similar messages and only one of them will be processed.

2.2.7. Randomness

One of our major concerns in our report from last year were the vulnerabilities resulting from potential attacks against the entropy source. Both the sever-side components and the voting clients are affected by this problem. In most cases, a successful attack would primarily affect vote privacy, but in case of the setup component, an adversary controlling the entropy seed of the pseudo-random number generator (PRG) may be able to change

the election result without being noticed. Our discussion of this problem in [HKLD22c, Section 2.7] was sufficiently profound, so we don't want to repeat ourselves here.

Our discussion of this topic also included several recommendations for improvements, but unfortunately, none of them has been implemented. The following statement from the response document indicates that there might be some misconception:

> *"There is a well-known best practice in cryptography called "do not roll your own crypto", stating that one should rely on standard, well-tested cryptographic methods instead of developing custom ones. [...] Cryptographic pseudorandom number generators are available in most programming languages and rely on the operating system to provide high-quality entropy sources. Building a custom entropy collector and pseudorandom number generator would violate the above-stated principle, and we prefer not to take any unnecessary risks."* [ResScope2, Section 4.3]

We agree that one should not develop alternatives for standard cryptographic methods, but this is not what we asked. The point is that a seed from a poor entropy source can not cause any damage when using it for seeding or re-seeding a PRG, as long as at least one high-entropy seed from another source has been available before or is used simultaneously. In other words, it is always better to use as many entropy sources as possible, even if some of them are poorly implemented.

At the moment, however, Swiss Post still delegates the whole security of the PRG to one single entropy source, both on the server and on the client, and not even runs a health test to detect the simplest types of failures. From all the remaining problems listed in this report, which think that this is the most critical one. Especially on the client, where a single line of infiltrated JavaScript code can render the standard PRG from the Web Crypto API useless (see example discussed in [HKLD22c, Section 2.7]), the vulnerability seems relatively easy to exploit, for example by attacking one of the numerous direct or indirect external dependencies. Note that the proposed counter-measures of keeping a reference to the PRG function in a private scope would be relatively easy to implement.

3. Systematic Analysis

To evaluate the current implementation in the light of the manifold improvements and corrections made since the 2021 version of the system, we conducted an even more systematic analysis of both the protocol description and the available Java, JavaScript, and TypeScript code. In the center of our attention were the algorithms as defined in [CryptPrim], [SysSpec], and [VerSpec], and corresponding classes and methods found in the code base. An obvious improvement compared to earlier versions is the adoption of exactly the same algorithm names in the code. With respect to this particular aspect, the alignment level reached in the current version is almost perfect. This advancement is very helpful for efficiently locating the code lines in the code base that are responsible for implementing a given algorithm.

The main content of this section is a systematic summary of all our findings and recommendations relative to all algorithms specified in the available documents. Our presentation of this summary is structured according to the available software components: Subsection 3.2 analyses the crypto-primitives component, Subsection 3.3 the e-voting component, and Subsection 3.4 the verifier component. In Subsection 3.1, we discuss some general problems that we have encountered either in the specification or throughout the code base. In each case, we provide recommendations for making corresponding improvements.

Given the large number of problems found in all areas of both the specification and the code, we believe that an additional round of carefully addressing all the points raised in this section will be necessary to reach a satisfactory degree of documentation and code quality. Note that the vast majority of the listed issues are not critical, and it should be possible to fix or improve them easily given our comments and recommendations. In the summaries given in Subsections 3.2 to 3.4, algorithms containing such non-critical issues are marked in orange. There are also some algorithms containing critical issues that are marked in red, for example if the desired matching between code and specification is clearly violated. From our perspective, the improvement of these algorithms is mandatory. Finally, for algorithms marked in green, we have no recommendations for improvements.

Note that we used the above colors to highlight the results of analyzing the August release. Many of the problems encountered have been resolved in the November or December releases (see Appendix B). In such cases, we added comments of the form [updated in November release] or [updated in December release] to the text, but we did not change the colors. Therefore, except in cases where additional text has been added to these comments, the colors do not necessarily reflect the project state at the end of this assessment.

3.1. General Problems

In this subsection, we describe some general problematical points in the current version of the protocol and the system, which can not be attributed to a single algorithm or a specific system component. Some of the raised issues are specific to either Scope 1 or Scope 2, but most of them affect both scopes. We believe that these issues should be addressed for improving future versions of the system, even if they are possibly not critical for the current system's overall security.

3.1.1. Distinction between CCR and CCM

The specification still makes a relatively explicit distinction between two groups of control components, *return codes control components* (CCR) and *mixing control components* (CCM). While the CCRs are active during the setup and voting phase [SysSpec, Fig. 6, 8, 9], the CCMs are active during the setup and the tally phase [SysSpec, Fig. 7, 10, 11]. In the current version of the specification document, the distinction between the CCRs and CCMs is defined as two different *functionalities* of a single entity:

> *"Conceptually, we distinguish two control component functionalities: [...] In the specification, we refer to each functionality separately, even though the control components are a single entity combining the CCR and CCM functionalities."*

This statement on Page 15 of [SysSpec] is the only explanation that we found, which expresses the fact that the CCRs and CCMs are no longer considered as distinct protocol parties. In earlier version of the protocol, the distinction was always strict and explicit.

The definition of the CCRs and CCMs in the current protocol version as two functionalities of the same party is quite confusing. For example, it is not clear how the main input parameter $\mathbf{vcMap}_j$ of the algorithm GetMixnetInitialCiphertexts is transferred from the CCRs to the CCMs without considering them as a single protocol party. Also, in a setting in which they are considered as distinct parties, it is not clear whether this is compatible with the [OEV] trust model of at least one trustworthy control component in a group of four control components.

The point here is that adhering to the terms CCM and CCR does not seem to generate an obvious benefit in the current version of the protocol specification. We understand that historically these terms were very important in the design of the protocol and the system, but since this is no longer the case in the current version, we recommend eliminating them entirely (replace CCR and CCM everywhere by CC).

3.1.2. Definition and Usage of Context

In principle, the separation of algorithm parameters into *context variables* and *input variables* is a very good design strategy to cope with the fact, that certain values remain fixed over multiple invocations of the algorithm and others are different for each invocation. A statement for describing this general guideline is included in all three specification documents under "*Conventions*":

> *"We designate values that do not change between runs as Context and variable values as Input."* [CryptPrim, Section 1.1]

> *"Distinguish context (invariant) and input variables (different for each invocation)."* [SysSpec, Section 1.4], [VerSpec, Section 1.2]

Unfortunately, there is no clear definition of *the context* in any of the available specification documents. By looking at the context variables of all algorithms, it is difficult to recognize a clear and consistent plan of the context's boundaries. To obtain a complete overview, we compiled all context variables into one big list and sorted them according to different categories. The tables shown below for each category are the results of this compilation:

Algorithms and Algorithm Parameters
– Cryptographic hash function Hash
– Extendable output function XOF
– Authenticated encryption function AuthenticatedEncryption
– Authenticated decryption function AuthenticatedDecryption
– The signature algorithms, providing GenKeyPair and GetCertificate
– The signature algorithms, providing Verify
– The trust store, providing FindCertificate
– Argon2: Memory usage parameter m
– Argon2: Parallelism parameter p
– Argon2: Iteration count i

Security and Group Parameters
– The security level λ
– Group modulus p
– Group cardinality q
– Group generator g

– A commitment key **ck**

Length Parameters

– Identifier length $\mathtt{l}_{\mathtt{ID}}$
– Character length of the Ballot Casting Key $\mathtt{l}_{\mathtt{BCK}}$
– Character length of the Base64 encoded hash output $\mathtt{l}_{\mathtt{HB64}}$
– Character length of the Start Voting Key $\mathtt{l}_{\mathtt{SVK}}$
– Character length of the Choice Return Codes $\mathtt{L}_{\mathtt{CC}}$
– Character length of the Vote Cast Return Code $\mathtt{L}_{\mathtt{VCC}}$

Identifiers and Indices

– Election Event ID `ee`
– Ballot box ID `bb`
– Verification card set ID `vcs`
– Vector of verification card set IDs **vcs**
– The CCR's index j
– The other CCR's indices $\hat{\mathbf{j}}$

Election Parameters

– Number of voting options n
– Number of selectable voting options ψ
– Number of allowed write-ins + 1 for this specific ballot box $\tilde{\delta}$
– Maximum number of possible voting options ω
– Maximum number of selectable voting options φ
– Maximum number of supported write-ins + 1: μ
– Actual voting options $\tilde{\mathbf{v}}$
– Encoded voting options $\tilde{\mathbf{p}}$
– Correctness IDs of all voting options **ciVotingOptions**
– Correctness IDs of the selected voting options **ciSelections**
– Number or voters $\mathtt{N}_{\mathtt{E}}$

Keys and Certificates

– Setup secret key $\mathtt{sk}_{\mathtt{setup}}$
– Setup public key $\mathtt{pk}_{\mathtt{setup}}$
– Election public key $\mathtt{EL}_{\mathtt{pk}}$
– A multi-recipient public key **pk**

– Choice Return Codes encryption public key $\mathbf{pk}_{\mathrm{CCR}}$
– The private key privKey
– The matching certificate cert

Lists and Maps

– List of generated voting cards $\mathsf{L}_{\mathtt{genVC},j}$
– List of voting cards with decrypted partial Choice Return Codes $\mathsf{L}_{\mathtt{decPCC},j}$
– List of voting cards $\mathsf{L}_{\mathtt{sentVotes},j}$
– List of confirmed voting cards $\mathsf{L}_{\mathtt{confirmedVotes},j}$
– Key-value map of confirmation attempts per verification card $\mathsf{L}_{\mathtt{confirmationAttempts},j}$
– List of shuffled and decrypted ballot boxes $\mathsf{L}_{\mathtt{bb},j}$
– List of Partial Choice Return Codes $\mathsf{L}_{\mathtt{pCC}}$
– List of shuffled and decrypted ballot boxes $\mathsf{L}_{\mathtt{bb},\mathtt{Tally}}$

The above compilation of the context variables shows the complexity of this topic, reveals some problematical cases, and exhibits a potential for simplifications. The first category *Algorithms and Algorithm Parameters*, for example, is something that could be defined outside the context, for example in a special section of the specification document, which discusses and defines the selection of the cryptographic primitives. Then we observe that the second and third categories, *Security and Group Parameters* and *Length Parameters*, define the system's security, whereas the next two categories, *Identifiers and Indices* and *Election Parameters*, define the current election. Thus, instead of having a single large context, it would obviously make sense to distinguish between a *security context* and an *election context*. For each algorithm, one could then simply define which of the two contexts is required, without giving the whole list of context variables over and over again. The potential for simplifications from introducing this simple distinction is worth taking into account.

A problematical type of context variables are the ones listed under *Keys and Certificates*. In cryptographic schemes, keys are usually important parameters of all fundamental algorithms. It is true that for a given party, some keys may not change during an election, but since keys are usually attributed to a single party, they should not be part of a (possibly common) context. The most obvious cases of doubtful context variables are the setup secret key $\mathtt{sk}_{\mathtt{setup}}$ in CombineEncLongCodeShares and GenCMTable and the private key privKey in GenSignature. From our point of view, secret or private keys should never be included in a context.

Even more problematical are the context variables listed under *Lists and Maps*, since there are cases, in which these lists are modified during the execution of the algorithm. For example in PartialDecryptPCC, elements are added to the list $\mathsf{L}_{\mathtt{decPCC},j}$ included in the context. Conceptually, context variables should be *immutable* by definition, but this is not the case in the given example. Generally, we recommend excluding data structures

such as lists or maps from any context to ensure that all algorithms are free from side-effects.

Another problem related to the context variables is the fact that certain variables are sometimes included in the context and sometimes in the regular list of input parameters. Probably the best example of that kind are the group parameters p, q, and g. In `GenKeyPair`, for example, they are defined as ordinary input parameters, but in GetCiphertext (and in many other algorithms), they are defined as context variables. Examples of that kind can be found at many different locations. They demonstrate that the concept of a context is not implemented in a rigorous and consistent manner, both in the specification and the code.

A general problem with any context is the question of how to authenticate the values of its variables. In our proposal from above, the security context could be initialized based on a public seed and security level, whereas the election context could be signed by an election administrator at the beginning of the setup phase. These are examples of the two most obvious ways for the parties to achieve sufficient trust in the context variables. Unfortunately, this important discussion is entirely missing in the current specification documents. In [SysSpec, Section 3.3], for example, the definition of the *election event context* says nothing about its distribution to the parties during the setup phase.

In the implementation of the algorithms, the context is not always handled in a clear and consistent way. Without knowing the code, we would have expected to find a class `Context` (or multiple classes such as `ElectionContext` or `SecurityContext`) somewhere in the code, such that instances of these classes can be kept by each party for passing them as additional inputs to the algorithms. Unfortunately, this is not the case in the current implementation. The closest abstractions of that kind are the classes `ReturnCodesNodeContext` and `VerificationCardSetContext` in the e-voting component, which consist of instance variables textttelectionEventId, `verificationCardSetId`, `ballotBoxId`, `numberOfWriteInFields`, and `numberOfVotingCards` for the election parameters `ee`, `vcs`, `bb`, $\tilde{\delta}$, and $|\mathbf{vcs}|$, respectively, and a reference to a `GqGroup` instance `encryptionGroup`.

In many cases, the context variables are passed as an additional parameter to the method implementing the algorithm. For example, `HashService::hashAndSquare` defines an additional argument of type `GqGroup`, which provides the context variables p and q from Algorithm 4.11. In other cases, the context variables are available as instance variables in corresponding utility or service classes. For example, the method `EncryptionParameters::getEncryptionParameters` obtains the context variable λ from an instance variable `lambda` of type `SecurityLevelInternal`, which itself stores the number of security bits. Sometimes, context variables are derived from the actual parameters. The method `FactorizeService::factorize`, for example, derives the group parameters from the first argument `x` of type `GqElement`, which itself holds a reference to an instance of `GqGroup` providing the values p, q, and g. Finally, there are cases in which context variables (mainly the lists used during the voting phase) are indirectly accessed using a service during the algorithm. For example, `PartialDecryptPCCAlgorithm::partialDecryptPCC`

uses an instance of the service class `VerificationCardStateService` to perform the operations related to the list $\mathsf{L}_{\mathtt{decPCC},j}$.

The point of mentioning the above examples is to demonstrate that a clear design pattern for implementing the context is completely missing in both the specification and the code. To improve the overall documentation and code quality, we recommend redesigning and implementing this entire topic from scratch.

3.1.3. Election Result

At the very end of the whole election process, by calling ProcessPlaintexts, the tally control component computes the election result by factorizing the decrypted votes. Therefore, the result of this algorithm is a *list of all selected encoded voting options* $L_{\mathtt{votes}} = (\hat{\mathbf{p}}_0, \ldots, \hat{\mathbf{p}}_{\mathtt{N_C}-1})$ containing the factorizations $\hat{\mathbf{p}}_i = (\hat{p}_{i,1}, \ldots, \hat{p}_{i,\psi-1})$, $\hat{p}_{ij} \in \mathbb{G}_q \cap \mathbb{P}$, for each of the $\mathtt{N_C}$ submitted and confirmed votes. Without giving more explanations, $L_{\mathtt{votes}}$ is implicitly defined as the *election result*. The problem is that $L_{\mathtt{votes}}$ alone has no semantics, i.e., it does actually not define the election result, only its encoding as prime numbers. What is missing is the so-called pTable, which is defined in [SysSpec, Section 3.4.2] as the combination of the context variables $\tilde{\mathbf{v}}$ (actual voting options) and $\tilde{\mathbf{p}}$ (encoded voting options). Note that the elements $v_i \in \tilde{\mathbf{v}}$ are strings describing or identifying the voting options. Something else that is entirely missing in the election result are the decrypted write-ins (ProcessPlaintexts ignored them).

The general problem here is that a clear definition of the *election result* is missing in the current specification document. This is very unfortunate, because the election result is clearly the main outcome of executing the protocol. The current situation, in which the election result implicitly consists of two objects, a list $L_{\mathtt{votes}}$ and a map pTable betweeen two vectors $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{p}}$, is not very satisfactory, because it contains irrelevant information (the internal vote encoding as prime numbers), requires additional computational steps (applying pTable to $L_{\mathtt{votes}}$), and completely ignores important information from the submitted ballots (the decrypted write-in options).

From the perspective of the verifier, it is very important to verify not only the correctness of the decrypted votes in $L_{\mathtt{votes}}$, but also the authenticity of pTable, which originates from the setup component. Therefore, verifying an election must always involve checking the setup component's signature, because otherwise the election result could be arbitrarily modified using a corrupt pTable. This is clearly a very critical point, but it could be entirely avoided using a more sophisticated definition of the election result. [updated in October release]

Probably the simplest solution would be to define a unique enumeration of the voting options, for example based on the lexicographical ordering of their identifiers from the official eCH document specifying the election. Since $\tilde{\mathbf{p}}$ is obtained from GetSmallPrimeGroupMembers in a deterministic manner for a given group $\mathbb{G}_q$, it would then be possible to define the vote encoding without explicitly introducing the pTable map. The

election result could thus be defined independently of pTable, simply by mapping the prime numbers back to the unique enumeration of the voting options. This would then be the last step of the ProcessPlaintexts algorithm. We recommend implementing this solution to eliminate current obstacles and to improve the clarity of the protocol's ultimate outcome.

3.1.4. Voting Server and Authentication

The voting server is an untrusted party, which mainly serves as a communication hub between the other protocol parties. There are two exceptions during the voting phase, where the voting server first executes ExtractCRC in [SysSpec, Figure 8] and then ExtractVCC in [SysSpec, Figure 9]. This corresponds to the definition of the voting server's role as defined in the specification document:

> *"The voting server relays messages to the control components and extracts the short return codes."* [SysSpec, Section 2.3]

Since the voting server is untrusted, there is no guarantee for the voting client (and the voter) that the return codes $\mathbf{CC}_{\mathtt{id}}$ and $\mathtt{VCC}_{\mathtt{id}}$ have been computed correctly. Alternatively, the protocol could let the voting client execute ExtractCRC and ExtractVCC directly on the inputs of the control components. This would reduce the voting server's role into a pure communication hub with no additional responsibilities. For making the presentation of the protocol more compact, the voting server could be removed completely from the discussion and the protocol diagrams in [SysSpec], which as a side-effect would further emphasize the logical information flow of the protocol. We believe that this tiny change in the voting server's role would greatly simplify the overall presentation of the protocol.

Another problem related to the voting server' role is the authentication of the messages sent to the voting server or received from it. Here are a few quotes from the specification document, which state that authentication is important in both directions, but without further specifying its implementation:

> *"The voters authenticate to the voting server"* [SysSpec, Section 2.1]

> *"We omit the voter's authentication to the voting server to retrieve the Verification Card Keystore [...] and we assume that the voting client authenticates to the voting server prior to the SendVote phase."* [SysSpec, Section 5.1]

> *"However, the elements exchanged between the control components and the voting server should still be authenticated [...]."* [SysSpec, Section 7]

From the summaries of the signed messages in [SysSpec, Table 15 and 16], we conclude that the voting server issues signatures for its outgoing messages, and this implies that it must possess a signature key pair and a certificate known to the control components. However, the voting server's keys and certificate are not further specified.

From a protocol design perspective, we do not see an obvious reason for imposing the voting server to sign its outgoing messages. For example, it is not important for the voting client whether a message from a control component has passed over the voting server or not, as long as it is clear that the massage originates from the control component. Eliminating the voting server from the protocol, as suggested above, would make the sender and receiver of all authenticated channels more visible. Given that the voting server is untrusted, the value of an issued signature is questionable anyway.

3.1.5. Minor Problems

We conclude this subsection of general problems encountered during our thorough analysis of the specification and the code by a list of independent minor issues.

- The assumptions described in [CryptPrim, Section 6.2] about importing the certificates are not at the right place. They should be moved to [SysSpec].
- We recommend removing the index j from algorithm names like in $\texttt{MixDecOnline}_{\texttt{j}}$, because an algorithm exists independently of its caller.
- The pseudocode uses the symbols $\top$ and $\bot$ for the truth values *true* and $false$, respectively. However, sometimes the statement **return** $\bot$ means to abort the algorithm in an exceptional case, for example in Line 12 of GetSmallPrimeGroupMembers. These cases are usually implemented by throwing an exceptions, sometimes using methods such as `Preconditions::checkState` from the Google's Guava library. From the point of view of the pseudocode algorithms, it is confusing to use the same symbols for very different purposes.
- The convention to "*display [...] vectors in boldface*" is introduced in [SysSpec, Section 1.4], but it is violated in many places in [CryptPrim, arg1], for example in GetCiphertextVectorExponentiation, `GetShuffleArgument`, VerifyShuffleArgument, and corresponding sub-algorithms.
- Proper doc documentation is missing at many places (`mvn javadoc:javadoc` outputs many warnings).

3.2. Cryptographic Primitives

The crypto-primitives component, which consists of a total of 92 well-specified pseudocode algorithms, is responsible for all cryptographic computations during a protocol run. As such, this component is clearly the centerpiece of the whole system and most critical for

the system's security. Carefully analyzing this component has therefore been a key task in our evaluation, and we have given it a lot of attention.

With a few exceptions, the algorithms of this component can easily be located in the code using their names as a search term over the project files. Most algorithms are specified in a Java interface and implemented in a service class implementing the interface. For example, RecursiveHash (Algorithm 4.8) is specified as a method `recursiveHash` in an interface called `Hash` and implemented in a class called `HashService`. This pattern is applied throughout the whole code base of the crypto-primitives component. The consequence of this design pattern is the fact that the component's algorithms are spread across a large number of files in many different packages, which possibly makes keeping an overview harder than necessary.

3.2.1. General Problems

In our analysis of the algorithms and their implementations, we made some general observations that we would like to discuss first. Each of the issues raised in the following list could be a useful pointer to a potential improvement of either the specification or the code. The same holds for the large number of specific issues raised in the following subsections. Generally, we recommend performing an additional round of carefully fixing all the encountered problems of this component for further improving its quality. Like in any cryptographic implementation, achieving the highest possible level of clean coding and documentation should be a key objective for this centerpiece component.

Stream Programming

Stream programming is undoubtedly a great enhancement of Java since its introduction in Java 8. It can be used to replace traditional loops by a more compact, concise, and powerful pipeline notation. From this point of view, we understand that stream programming has been used all over the code base of the crypto-primitives component. However, given the fact that all pseudocode algorithms are specified using traditional for- and while-loops, this may create a problem for people evaluating the code that are not familiar with stream programming. To illustrate the problem with an example, we show below the loop in Lines 4–6 of the algorithm GenDecryptionProof together with the corresponding Java stream programming code:

4: **for** $i \in [0, \ell)$ **do**
5: $\quad y_i \leftarrow \mathsf{pk}_i$
6: $\quad y_{\ell+i} \leftarrow \frac{\phi_i}{m_i}$

```
final GroupVector<GqElement, GqGroup> y = Stream.concat(
                pk.stream().limit(l),
                IntStream.range(0, l).mapToObj(i -> phi.get(i).multiply(m.get(i).invert())))
        .collect(toGroupVector());
```

As the above example demonstrates, stream programming may not always be the best choice for creating easily readable code. To further improve the matching between pseudocode and Java algorithms, we recommend implementing all pseudocode loops in a one-to-one manner by traditional Java loops, and to use stream programming wherever it is useful in helper methods and utility classes, or to invoke parallel computations. Note that we already recommended to use stream programming more defensively in our report from last year [HKLD22c, Section 2.2.1]. We also want to remind the developers of their own statement about using streams from the architecture document:

> "*However, we use for-loops when it leads to a better alignment with the pseudo-code algorithms and when the computational advantage is negligible.*" [ArchDoc, Section 9.2]

Limiting Cases

We have encountered many examples in which limiting cases are explicitly excluded, for example by $N > 0$ (at least one ciphertext), $N \geqslant 2$ (at least two ciphertexts to shuffle), $\ell > 0$ (at least one multi-recipient ElGamal message or ciphertext), $\nu \in \mathbb{N}^+$ (at least one commitment key), $n \in \mathbb{N}^+$ (at least one random integer to generate), $n \in \mathbb{N}^+$ (a non-empty byte array), $l \in \mathbb{N}^+$ (at least one padding character), $k \in \mathbb{N}^+$ (a non-empty string), $\ell \in \mathbb{N}^+$ (length of random string), $l \in \mathbb{N}^+$ (desired code length), etc. Although there are also a few counter-examples, it seems that the exclusion of limiting cases is an algorithmic design decision.

In most of these cases, however, the algorithm would also work in exactly the same manner by allowing limiting cases as inputs (they would then for instance simply return an empty list). Therefore, we see this design decision as a violation of the generality principle in software engineering, according to which components should work as generally as possible for making them maximally robust and reusable. An example of a problem that arises from this violation is the shuffling of ciphertexts, which is restricted to an input list of size $N \geqslant 2$. In its application in the protocol, a supplementary preparation step within GetMixnetInitialCiphertexts is necessary only for dealing with this restriction (see Subsection 3.2.7 for further explanations). To avoid such problems, we recommend seeing the inclusion of limiting cases as an algorithmic design principle and to apply it wherever possible.

Parameters of Sub-Algorithms

In the current implementation of the specified algorithms, testing the domains of the input parameters is systematically applied across the whole crypto-primitives component (exceptional cases in which this is not perfectly the case are mentioned in the summaries given below). Since the code for performing these tests is sometimes longer and more complex than the code implementing the algorithm itself, it can be seen as a distracting factor that makes the code inspection more difficult. Nevertheless, it is clear that performing these tests is a matter of great importance for the robustness of the whole system.

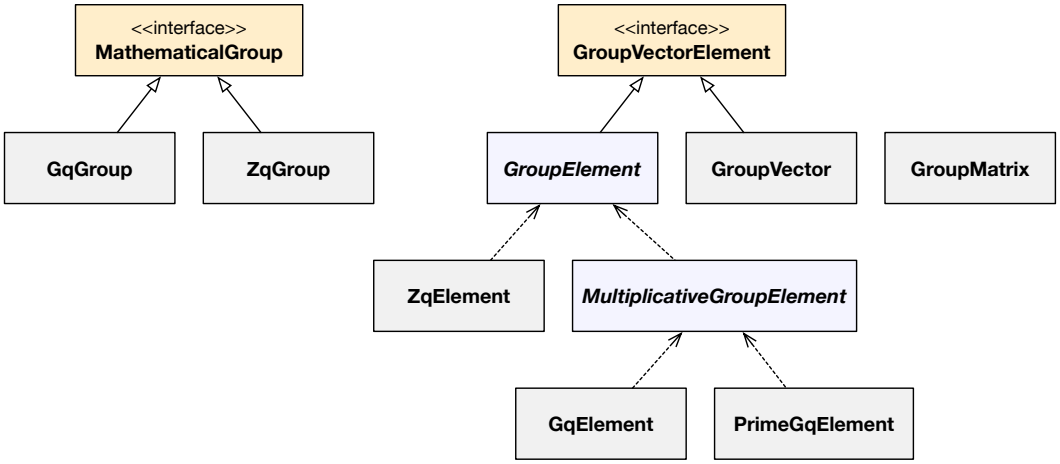
To improve this situation, we recommend removing these checks from all algo-

rithms that are only called as sub-algorithms within the crypto-primitives component. In this way, the responsibility of checking the domains of the parameters is delegated to the top-level algorithms, which are directly called by the protocol parties. In software engineering terms, this would correspond to implementing all top-level algorithms using *defensive programming* (all preconditions are strictly tested) and all sub-algorithms according to the *design by contract* principle (no preconditions are tested). As a consequence, implementations of sub-algorithms must then be regarded as something that should remain invisible to the outside world, for example by declaring corresponding Java methods as `private` or by excluding the packages of corresponding classes from the module-info.java file.

The potential for simplifications by this simple measure is considerably large. As an example, consider the implementation of the Bayer-Groth mix-net, which consists of two top-level algorithms GenVerifiableShuffle (Algorithm 8.1) and VerifyShuffle (Algorithm 8.2), and 24 strict sub-algorithms (Algorithm 8.3–8.26), some of them with a long and complex list of input parameters. Removing the code for testing the same parameters repeatedly in all these sub-algorithms would greatly improve the slimness of the mix-net implementation without any negative consequences.

Implemention of Mathematical Groups

The crypto-primitives component contains a package called `cryptoprimitives.math`, which provides various interfaces and classes for dealing with the mathematical groups $\mathbb{G}_q$ and $\mathbb{Z}_q$ and their elements. Since elements of these groups are used almost everywhere in the protocol, this package is quite fundamental. The following UML class diagram shows the available interfaces and classes and their relationships.



Generally, we believe that this OOP model is not yet an optimal abstraction of this fundamental subject. Here is a list of points that could be improved:

- The names of some classes and interfaces are misleading. `GroupVectorElement`, for instance, combines the terms of three different concepts (*group*, *element*, *vector*) in one. The name is therefore anything but self-explanatory. Other problematical names are `GroupVector` and `GroupMatrix`, which represent vec-

tors and matrices of *elements*, respectively, not vectors and matrices of *groups*.

- The class structure contains some asymmetries, for example `GqElement` inheriting from `MultiplicativeGroupElement`, but `ZqElement` not inheriting from `AdditiveGroupElement`. Another example is `GroupVector` implementing the interface `GroupVectorElement`, but `GroupMatrix` not implementing `GroupVectorElement`.
- Since $\mathbb{G}_q \cap \mathbb{P}$ is a subset of $\mathbb{G}_q$, it follows that every element of $\mathbb{G}_q \cap \mathbb{P}$ is also an element $\mathbb{G}_q$. Consequently, instances of `PrimeGqElement` should also be instances of `GqElement`, but with `PrimeGqElement` inheriting directly from `MultiplicativeGroupElement`, this is not the case. This is therefore a violation of the OOP specialization principle.
- Some of the interfaces do not or only partially provide the methods that one would expect. `MathematicalGroup`, for example, provides methods for obtaining the group size or the identity element, but not for computing the inverse or the group operation.

This particular code area with all the problems listed above is exemplary for other code areas, in which the current OOP design is not yet fully satisfactory. We recommend applying the highest possible degree of carefulness to further improve the quality of the OOP modeling and coding in such important areas.

Implementation of Tuples, Vectors, and Matrices

The mathematical description of the cryptographic protocol and the pseudocode algorithms heavily depend on *tuples*, *vectors*, and *matrices*. These concepts are fundamental for grouping related objects such as numbers, strings or byte arrays into composed objects, or even for doing so recursively to obtain arbitrary trees of such objects. Given their importance in all areas of the specification and the code, it is somewhat surprising that the crypto-primitives component does not provide a sufficiently general abstraction for dealing with such composed objects in the most convenient possible way.

The existing classes `GroupVector` and `GroupMatrix` are clearly insufficient as a general toolbox, because their application is restricted to vectors and matrices with values taken from $\mathbb{G}_q$. Given the lack of a more general toolbox for tuples, vectors, and matrices, we observe that they are often implemented with ordinary lists, i.e., using instances of classes implementing the generic Java interface `List`. Conceptually, this is clearly not the right instrument, because ordinary lists are mutable by definition, while vectors are immutable (see next topic). In the light of this remark, inheriting `GroupVector` from `ForwardingList` is clearly a conceptual mistake, because a vector is not a specialization of a list.

For greatly improving the code all across the entire code base, we highly recommend to equip the crypto-primitives component with generic implementations of tuples (pairs, triples, etc.), vectors, and matrices, and to strictly apply them in every possible situation. As an example of such an implementation, we refer to

corresponding classes in the OpenCHVote implementation.[6]

Interface Hashable

The implementation of the interface `Hashable` and its sub-interfaces `HashableString`, `HashableInteger`, `HashableByteArray`, and `HashableList` is another example of sub-optimal OOP modeling. The primary purpose of these interfaces is to restrict the possible input parameters of RecursiveHash to objects that can actually be hashed. Since hashing ultimately requires the input to be encoded as a byte array, one would expect from the interface `Hashable` to define a method `byte[] toByteArray()`, but this is unfortunately not the case (it only defines a method `Object toHashableForm()`, which essentially implements the unwrapping function from an instance of `Hashable` back to the original object). This implies that RecursiveHash needs to make its own distinction of cases using the `instanceof` operator, which is not very elegant. Therefore, `Hashable` currently makes the code unnecessarily complicated, because it implies a large number of wrapping operations in many places of the code (53 times `HashableBigInteger.from`, 22 times `HashableString.from`, 10 times `HashableList.from`), without generating a clear benefit. This could be avoided by either completely abandoning this idea or implementing it more carefully.

Immutability

Minimizing mutability is an important design principle for building robust, easy-to-test, and thread-safe software components [Blo18, Item 17]. The developers have applied this principle at many places of the crypto-primitives component (comments like "*Instances of this class are immutable*" appear in approximately 45 different classes). However, immutability is not always reached as claimed. Objects of the class `HashableByteArray`, for example, are mutable, as the following code example demonstrates (executing this code first outputs `[1, 2, 3]` and then `[-1, 2, 3]`):

```
public static void main(String[] args) {
    var hashableByteArray = HashableByteArray.from(new byte[]{1, 2, 3});
    System.out.println(Arrays.toString(hashableByteArray.toHashableForm()));
    hashableByteArray.toHashableForm()[0] = -1;
    System.out.println(Arrays.toString(hashableByteArray.toHashableForm()));
}
```

Interestingly, the mutability of `HashableByteArray` is exploited twice for wiping the content of a given password, first in `SetupTallyEBAlgorithm::setupTallyEB` (Line 135) and second in `MixDecOfflineAlgorithm::mixDecOffline` (Line 156). In the light of theses examples, it seems as if the mutability has been allowed on purpose for hashable byte arrays. Two other classes that are wrongly declared as immutable are `VerificationFailure` and `VerificationSuccess`. Other examples of insufficient implementation of immutability can be found in the e-voting component

[6] See module utilities at https://gitlab.com/openchvote/cryptographic-protocol.

(see Subsection 3.3).

With respect to dealing with byte arrays, it would have been much better to implement a regular class `ByteArray`, which is truly immutable, instead of writing a utility class `ByteArrays` for dealing with byte arrays of type `byte[]`, which are by definition not immutable. The same holds for tuples, vectors, and matrices (see discussion above), which also could have been implemented by truly immutable classes. In this way, the widespread application of the methods `List::of`, `List::copyOf`, and `Stream::toList` only to obtain unmodifiable lists could be dramatically reduced and simplified. Generally, the principle of *defensive copying* should be applied to the constructors of immutable classes, not to arrays or instances of mutable classes.

Package Structure

The package structure of the crypto-primitives component consists of 9 sub-packages with names such as `elgamal`, `hashing`, `math`, etc., plus an additional sub-package `internal`, which itself contains the same 9 sub-packages `elgamal`, `hashing`, `math`, etc. To the best of our understanding, this structure has been defined to allow the hiding of some classes in the module-info.java file. Nevertheless, we found it very confusing, because it often meant to find closely related interfaces and classes in two very different locations.

Another problem in the component's package structure is the name of the package `internal.securitylevel`, which contains implementations of cryptographic algorithm such as AES, SHA3, or SHAKE. The package name is therefore very confusing. Furthermore, the utility class `VectorUtils` is clearly in the wrong package, because it only deals with objects of type `GqElement`, `GqGroup`, or `GroupVector`, which are all located in the package `math`. These are just a few examples of minor problems, but the general point here is that the current organization of the packages is not optimal in every aspect.

3.2.2. Basic Data Types

A first fundamental category of algorithms deals with primitive data types such as strings, integers, and byte arrays and the conversion between them. Corresponding encoding and decoding algorithms are specified by referring to RFC4648 and RFC3629. Since implementations of these standards are widely available in most programming languages, this is undoubtedly a reasonable choice.

What is regrettable, however, is the fact that the specification includes wrapper algorithms for these standards (e.g., Base16Encode and Base16Decode), but in the implementation, corresponding wrapper methods are missing. Instead, methods from existing libraries `com.google.common.io.BaseEncoding` and `java.util.Base64.Encoder` are directly called at various locations in the code. Hence, when changes need to be implemented in future versions, it might be necessary to modify the code at each of these locations.

Our general recommendation therefore is to provide such wrapper methods and use them consistently throughout the code, as it is done in the pseudocode algorithms. This would also further expose the alignment between code and specification. [updated in November release]

Algorithm 3.1: CutToBitLength	
Called by:	–
Subalgorithm of:	GenRandomInteger, RecursiveHashOfLength, KDFToZq
General comments:	We recommend specifying the first parameter as $B \in \mathcal{B}^*$ with $N = \|B\|$ instead of $B \in \mathcal{B}^N$. Otherwise, it seems as if N is a fixed value defined in the context. Furthermore, there is no obvious reasons for excluding the limiting case $N = n = 0$.
Code comments:	*none*

Algorithm 3.2: Base16Encode	
Called by:	–
Subalgorithm of:	GenRandomBase16String
General comments:	Refers to RFC4648.
Code comments:	This algorithm is not implemented explicitly. The external encoder from the class `com.google.common.io.BaseEncoding` is used instead at each call. [updated in November release]

Algorithm 3.3: Base16Decode	
Called by:	–
Subalgorithm of:	–
General comments:	Refers to RFC4648. Since this algorithm is never used, we recommend removing it from the specification.
Code comments:	This algorithm is not implemented explicitly. [updated in November release]

Algorithm 3.4: Base32Encode	
Called by:	–
Subalgorithm of:	GenRandomBase32String
General comments:	Refers to RFC4648.
Code comments:	This algorithm is not implemented explicitly. The external encoder from the class `com.google.common.io.BaseEncoding` is used instead at each call. [updated in November release]

Algorithm 3.5: Base32Decode	
Called by:	–

Subalgorithm of:	–
General comments:	Refers to RFC4648. Since this algorithm is never used, we recommend removing it from the specification.
Code comments:	This algorithm is not implemented explicitly. [updated in November release]

Algorithm 3.6: Base64Encode

Called by:	–
Subalgorithm of:	GenRandomBase64String, GenVerDat, CombineEncLongCodeShares, GenCMTable, GenCredDat, GetKey, CreateLCCShare, ExtractCRC, CreateLVCCShare, VerifyLVCCHash, ExtractVCC
General comments:	Refers to RFC4648.
Code comments:	This algorithm is not implemented explicitly. The standard encoder from the class `java.util.Base64.Encoder` is used instead at each call. [updated in November release]

Algorithm 3.7: Base64Decode

Called by:	–
Subalgorithm of:	ExtractCRC, ExtractVCC
General comments:	Uses the Base64 alphabet, but the missing reference to RFC4648 should be added.
Code comments:	This algorithm is not implemented explicitly. The standard decoder from the class `java.util.Base64.Decoder` is used instead at each call. [updated in November release]

Algorithm 3.8: ByteArrayToInteger

Called by:	–
Subalgorithm of:	GenRandomInteger, RecursiveHashToZq, KDFToZq, GetEncryptionParameters, GetShuffleArgument, VerifyShuffleArgument, GetMultiExponentiationArgument, VerifyMultiExponentiationArgument, GetHadamardArgument, VerifyHadamardArgument, GetZeroArgument, VerifyZeroArgument, GetSingleValueProductArgument, VerifySingleValueProductArgument, GenSchnorrProof, VerifySchnorr, GenDecryptionProof, VerifyDecryption, GenExponentiationProof, VerifyExponentiation, GenPlaintextEqualityProof, VerifyPlaintextEquality, GetKey
General comments:	We recommend specifying the parameter as $B \in \mathcal{B}^*$ with $n = \|B\|$ instead of $B \in \mathcal{B}^n$. Otherwise, it seems as if n is a fixed value defined in the context.
Code comments:	The method `ConversionsInternal::byteArrayToInteger` uses the conversion method from the `BigInteger` class. It is therefore not a one-to-one implementation of the pseudocode and its correctness is not obvious.

Algorithm 3.9: IntegerToByteArray	
Called by:	–
Subalgorithm of:	RecursiveHash, RecursiveHashOfLength, GetEncryptionParameters, GenEncLongCodeShares, GenCredDat, CreateLCCShare, CreateLVCCShare
General comments:	There is no obvious reason for representing 0 as <0x00> instead of <> (empty byte array). This makes Algorithm 3.9 unnecessarily complicated (Line 2) without creating a clear benefit.
Code comments:	The method `ConversionsInternal::integerToByteArray` uses the conversion method from the `BigInteger` class. It is therefore not a one-to-one implementation of the pseudocode and its correctness is not obvious.

Algorithm 3.10: ByteLength	
Called by:	–
Subalgorithm of:	IntegerToByteArray, GenRandomInteger, KDFToZq
General comments:	It's confusing to use n for the given integer and b for its length. In previous algorithms, integers were denoted by x and byte lengths by n. We recommend renaming the variables accordingly. [updated in November release]
Code comments:	An actual implementation of this algorithm is missing. [updated in November release] In the methods `ConversionsInternal::integerToByteArray` and `RandomService::genRandomInteger`, most of the computation is delegated to the `BigInteger` class, and in `KDFService::KDFToZq`, the byte length is computed locally. In all three cases, this is not a one-to-one implementation of the pseudocode. [updated in November release: only for `KDFService::KDFToZq`]

Algorithm 3.11: StringToByteArray	
Called by:	–
Subalgorithm of:	RecursiveHash, RecursiveHashOfLength, KDF, GenCiphertextSymmetric, GetPlaintextSymmetric, GetEncryptionParameters, GenCMTable
General comments:	Refers to UTF-8 as defined in RFC3629.
Code comments:	Uses the standard Java class `java.nio.charset.StandardCharsets`.

Algorithm 3.12: ByteArrayToString	
Called by:	–
Subalgorithm of:	ExtractCRC, ExtractVCC

General comments:	Refers to UTF-8 as defined in RFC3629. We recommend specifying the parameter as $B \in \mathcal{B}^*$ with $n = \|B\|$ instead of $B \in \mathcal{B}^n$. Otherwise, it seems as if n is a fixed value defined in the context. Furthermore, the restriction $n \in \mathbb{N}^+$ (instead of $n \in \mathbb{N}$) seems to be an unintended mistake, because it excludes decoding an empty byte array correctly into an empty string.
Code comments:	Uses the standard Java classes `java.nio.charset.CharsetDecoder` and `java.nio.charset.StandardCharsets`. It includes a test for $n \in \mathbb{N}^+$.

Algorithm 3.13: StringToInteger

Called by:	–
Subalgorithm of:	GenVerDat, CreateConfirmMessage
General comments:	The subalgorithm Decimal is not specified and no reference is given.
Code comments:	Implements Decimal using the constructor from the `BigInteger` class. The check in Line 2 only covers the first character of the given string. The remaining characters are checked by catching the potential exception thrown by the constructor. This is obviously correct, but it is not a one-to-one implementation of the pseudocode. [updated in November release: in the newly introduced regular expression `DECIMAL_PATTERN`, the end-of-line symbol `$` is missing to enable the removal of the try-catch clause]

Algorithm 3.14: IntegerToString

Called by:	–
Subalgorithm of:	GenUniqueDecimalStrings, VerifyEncryptedPCCExponentiationProofsVerificationCardSet, VerifyEncryptedCKExponentiationProofsVerificationCardSet
General comments:	The subalgorithm Decimal^{-1} is not specified and no reference is given.
Code comments:	Uses the `toString` method from the `BigInteger` class. This is obviously correct, but it should be better commented.

Algorithm 3.15: LeftPad

Called by:	–
Subalgorithm of:	GenUniqueDecimalStrings
General comments:	We recommend specifying the first parameter as $S \in \mathbb{A}^*_{UCS}$ with $k = \|S\|$ instead of $S \in \mathbb{A}^k_{UCS}$. Otherwise, it seems as if k is a fixed value defined in the context. Furthermore, there is no obvious reasons for excluding the limiting cases $k = 0$ and $k = l = 0$. Finally, the requirement $k <= l$ should be properly typeset as $k \leqslant l$ and the vector $\mathbf{p} = (c, \ldots, c)$ should be defined as a string $P = \langle c, \ldots, c \rangle$. Currently, using p instead of $\mathbf{p}$ in the second code line is obviously incorrect. [updated in December release]

Code comments:	The implemented method `RandomService::leftPad` is defined as *package-private*, which means that it cannot be called outside of `cryptoprimitives.internal.math`. On the other hand, an equivalent method `StringUtils::leftPad` from `org.apache.commons.lang3` is called twice in the modules `secure-data-manager` and `commons`, respectively. We recommend defining `RandomService::leftPad` as *public* and using it exclusively.

3.2.3. Basic Algorithms

A second category of fundamental algorithms deals with generating random integers and strings, hashing of structured mathematical objects, deriving keys, and primality testing. The most striking observation here is the absence of code implementing the *enhanced Baillie-PSW* (EBPSW) primality test from Algorithm 4.15 and its sub-algorithms 4.15 to 4.21. Alternatively, `BigInteger::isProbablePrime` is called at different locations, which only implements the regular BPSW test according to ANSI X9.80. Although EBPSW strengthens BPSW at almost no additional computational cost, we don't see the benefit of including detailed pseudocode algorithms for something that is not even implemented in the given code base. Moreover, since probabilistic primality tests are only required in GetEncryptionParameters for finding suitable group parameters in a deterministic, verifiable manner, there is no need for implementing the most advanced algorithm for detecting pseudoprimes generated under adversarial control. We therefore recommend removing these algorithms from the specification (a short discussion of the topic seems sufficient).

Algorithm 4.0: RandomBytes	
Called by:	–
Subalgorithm of:	GenRandomInteger, GenRandomBase16String, GenRandomBase32String, GenRandomBase64String, GenCiphertextSymmetric,
General comments:	Despite its critical role for the security of the protocol, this ultimately fundamental algorithm is not further specified (it has not even a proper algorithm number). The given statement that standard implementations for generating cryptographically secure randomness are available in most programming languages is certainly true, but given the delicacy of this topic, we would at least expect some recommendations about selecting an appropriate PRNG in practice. Clearly, the responsibility for this selection cannot be left to the software development team, nor should it be simply delegated to a programming language SDK at runtime.

Code comments:	The implementation in `RandomService::randomBytes` delegates the randomness generation to the standard Java class `java.security.SecureRandom` using its default constructor. In this way, the selection of the actual PRNG is determined at runtime by the operating system and thus remains undefined.

Algorithm 4.1: GenRandomInteger

Called by:	–
Subalgorithm of:	GenRandomVector, GenUniqueDecimalStrings, PassesMillerRabin, GenKeyPair, GenShuffle, GenShuffle, GetMultiExponentiationArgument, GetProductArgument, GetZeroArgument, GetSingleValueProductArgument, GenSchnorrProof, GenExponentiationProof, GenKeysCCR, GenVerDat, CreateVote
General comments:	For improved readability, we recommend replacing the Goto statement in Line 6 by either a while or a do-while loop. [updated in November release]
Code comments:	By delegating the generation of random integers to the constructor of the `BigInteger` class, the code is very different from the pseudocode. The Goto statement is implemented using a do-while loop.

Algorithm 4.2: GenRandomVector

Called by:	–
Subalgorithm of:	GetShuffleArgument, GetMultiExponentiationArgument, GetHadamardArgument, GetZeroArgument, GetSingleValueProductArgument, GenDecryptionProof, GenPlaintextEqualityProof
General comments:	There is no obvious reason for excluding the limiting case $n = 0$.
Code comments:	Implemented using stream programming.

Algorithm 4.3: GenRandomBase16String

Called by:	–
Subalgorithm of:	GenVerDat
General comments:	There is no obvious reason for excluding the limiting case $\ell = 0$. Furthermore, by calling three sub-algorithms, the algorithm is unnecessarily complicated. It would be much simpler to have a main loop over $i \in [0, \ell)$, to pick at each iteration a random index from $j \in [0, 16)$, and to select the j-th character from $\mathbb{A}_{Base16}$. The sequence of these characters is the random Base16 string.
Code comments:	Uses the external encoder from the `com.google.common.io.BaseEncoding` class. This is obviously correct, but it is not aligned with the pseudocode. To avoid code duplication, we recommend implementing the algorithms Base16Encode and Truncate and call them here. [updated in November release]

Algorithm 4.4: GenRandomBase32String

Called by:	–
Subalgorithm of:	GenVerDat
General comments:	Same remarks as for GenRandomBase16String.
Code comments:	Uses the external encoder from the `com.google.common.io.BaseEncoding` class. This is obviously correct, but it is not aligned with the pseudocode. To avoid code duplication, we recommend implementing the algorithms Base32Encode and Truncate and call them here. [updated in November release]

Algorithm 4.5: GenRandomBase64String

Called by:	–
Subalgorithm of:	–
General comments:	Same remarks as for GenRandomBase16String. Furthermore, since this algorithm is never used, we recommend removing it from the specification and the code base.
Code comments:	Uses the standard encoder from the `java.util.Base64` class. This is obviously correct, but it is not aligned with the pseudocode. [updated in November release] Since the algorithm is never used, the method is only called in test files.

Algorithm 4.6: Truncate

Called by:	–
Subalgorithm of:	GenRandomBase16String, GenRandomBase32String, GenRandomBase64String
General comments:	We recommend specifying the first parameter as $S \in \mathbb{A}_x^*$ with $u = \|S\|$ instead of $S \in \mathbb{A}_x^u$. Otherwise, it seems as if u is a fixed value defined in the context. Furthermore, there is no obvious reason for excluding the limiting cases $\ell = 0$ and $u = \ell = 0$. Finally, the output value S' should be specified as $S' = \langle S'_0, \ldots, S'_{\ell-1} \rangle$. Note that by simplifying Algorithms 4.3 to 4.5 as explained above, Truncate is no longer needed and can be removed from the specification.
Code comments:	This algorithm is not implemented explicitly. The standard method `String::substring` is used instead at each call. [updated in November release]

Algorithm 4.7: GenUniqueDecimalStrings

Called by:	–
Subalgorithm of:	GenVerDat, GenCMTable

General comments:	For the assignment of c, the proper assignment symbol $\leftarrow$ should be used. There is no obvious reason for excluding the limiting cases $n = 0$ and $l = n = 0$. Here again, the generation of the random string within the main while loop could be simplified by selecting l random digits instead of generating a random integer and representing it as a string with leading zeros. The algorithm LeftPad is then no longer needed.
Code comments:	*none*

Algorithm 4.8: RecursiveHash

Called by:	–
Subalgorithm of:	GenSignature, VerifySignature, GetVerifiableCommitmentKey, GetShuffleArgument, VerifyShuffleArgument, GetMultiExponentiationArgument, VerifyMultiExponentiationArgument, GetHadamardArgument, VerifyHadamardArgument, GetZeroArgument, VerifyZeroArgument, GetSingleValueProductArgument, VerifySingleValueProductArgument, GenSchnorrProof, VerifySchnorr, GenDecryptionProof, VerifyDecryption, GenExponentiationProof, VerifyExponentiation, GenPlaintextEqualityProof, VerifyPlaintextEquality, GenVerDat, CombineEncLongCodeShares, GenCMTable, GenCredDat, GetKey, CreateLCCShare, ExtractCRC, CreateLVCCShare, VerifyLVCCHash, ExtractVCC
General comments:	L is defined in the context as a positive natural number, so the requirement $L > 0$ is ambiguous. In Line 10, the correct test is $w \in \mathbb{A}^*_{UCS}$. There is no obvious reason for excluding the hashing of empty vectors in Line 12. Therefore, we recommend considering vectors $(w_0, \ldots, w_{j-1})$ of length $j \geqslant 0$. [updated in October release] The way vectors of length 1 (case $j = 0$ in Line 13) are handled as a special case creates trivial collisions between single values and vectors consisting of single values. We recommend removing Lines 13 and 14. Furthermore, we recommend adding an additional prefix byte (for example <0x03>) to the vector case to avoid trivial collisions between vectors and corresponding byte arrays obtained from applying RecursiveHash. Given the possibility of constructing such trivial collisions, the current algorithm is clearly not collision-resistant across the full input domain.[7] [updated in October release]
Code comments:	The code includes a prefix byte `ARRAY_PREFIX` for the vector case and the singleton case is not treated as a special case. Here, it seems that the specification is not up-to-date. [updated in October release]

Algorithm 4.9: RecursiveHashToZq

Called by:	–
Subalgorithm of:	HashAndSquare, GetVerifiableCommitmentKey, SetupTallyEB, MixDecOffline

General comments:	There is no obvious reason for the second requirement $\|q\| \geqslant 512$. In Line 3, applying the concatenation symbol in $h\|\|\mathbf{v}$ to an integer and a vector is a slightly abusive notation.
Code comments:	The code for prepending a single value to a list using stream programming is correct, but unnecessarily complicated. The problem here is the absence of proper data structures for dealing with mathematical objects such as vectors or tuples.

Algorithm 4.10: RecursiveHashOfLength

Called by:	–
Subalgorithm of:	RecursiveHashToZq
General comments:	Section number is missing in the reference given under **Require**. The variable ℓ^* in the second requirement is undefined. [updated in December release] It does not make sense to state $u \in \mathbb{N}^+ \times \mathcal{B}^* \rightarrow \mathcal{B}^u$. [updated in December release: but it is still incorrect, the correct statement would be XOF : $\mathbb{N}^+ \times \mathcal{B}^* \rightarrow \mathcal{B}^u$] Moreover, all remarks from RecursiveHash are also applicable here. [updated in October release]
Code comments:	An external library BouncyCastle is used for SHAKE-256 including XOF.

Algorithm 4.11: HashAndSquare

Called by:	–
Subalgorithm of:	GenVerDat, CreateLCCShare, CreateConfirmMessage, CreateLVCCShare
General comments:	The name of the algorithms is a bit misleading and inconsistent with other algorithm name. We recommend calling it RecursiveHashToGq, similar to RecursiveHashToZq, because it essentially defines a hash function into $\mathbb{G}_q$. Furthermore, there is no obvious reason for restricting the input to a single integer $x \in \mathbb{N}$. We recommend allowing multiple inputs $v_0, \ldots, v_{k-1}$ without restrictions, similar to RecursiveHashToZq. Note that in RecursiveHashToZq, the value q is defined as an input parameter, whereas in HashAndSquare, p and q are part of the context. We recommend doing it consistently in both cases.
Code comments:	The above remark about the algorithm's sub-optimal name becomes obvious by looking at the interface Hash. The specification defines p and q as part of the context, but the implementation has a single additional argument `group`, from which p and q are derived. Calling `GqElementFactory::fromSquareRoot` to compute modular squaring is correct, but not very obvious.

Algorithm 4.12: KDF

Called by:	–
Subalgorithm of:	KDFToZq, GenCMTable, ExtractCRC, ExtractVCC

General comments:	Concatenating the byte array representations of the contextual information may create unintended "collisions". [updated in October release] The call of the external algorithm HKDF-Expand uses the wrong font. [updated in December release]
Code comments:	By concatenating `info_i_bytes.length` to the byte array representations of each contextual information, the implementation in `KDFService::KDF` deviates from the pseudocode. Another deviation is the restriction `stringToByteArray(info_i).length <= 255` in Line 79, which is not present in the pseudocode. [updated in October release] Finally, making a copy of `contextInformation` is not necessary.
Algorithm 4.13: KDFToZq	
Called by:	–
Subalgorithm of:	GenEncLongCodeShares, CreateLCCShare, CreateLVCCShare
General comments:	Since Hash is not explicitly used (it is used only in the subalgorithm KDF), no restrictions on the variable L need to be imposed here.
Code comments:	The method `KDFService::KDFToZq` does not call `ByteLength` (which does not exist), but instead ℓ is computed locally. The test `l_curved >= L` is undefined in the pseudocode (since it involves L, it can simply be dropped).
Algorithm 4.14: Argon2id	
Called by:	–
Subalgorithm of:	GenCredDat, GetKey

General comments:	Switching from PBKDF2 to Argon2id for password based key derivation is a logical move that follows best practice in this problem domain. In contrast to PBKDF2 providing only one dimension to adjust the level of entropy gain, for Argon2id the specifier needs to consider three, namely p for handling the amount of lanes (CPUs), m for handling the amount of memory, and t for handling the amount of iterations. The specification claims the following values for providing "*a considerable increase in cost*": $p = 1$, $m = 14$, $t = 2$. The reasoning behind those specific values is neither explained nor supported by suitable material or methods better than "*. . . our benchmarks*". They are even in sharp contrast to the recent official RFC 9106, where $p = 4$, $m = 21$, $t = 1$ is provided as the first recommendation for any "uniform save option".[8] There are alternative recommendations for special cases, but in no case $p < 2$ is recommended.[9] There is even a paper indicating practical attacks on Argon2i and derivatives if $p = 1$ [AB17]. Therefore, we cannot support the reasoning behind the selected values. We suggest to provide concrete justification for the given setting. Besides, the generation of a random salt is missing, and it should be returned together with t. [updated in October release]
Code comments:	Both the name and the return values of the method `Argon2Service::genArgon2id` do not correspond to the pseudocode algorithm Argon2id. The auxiliary method `Argon2Service::getArgon2id` should be declared `private`. [updated in October release]

Algorithm 4.15: IsProbablePrime	
Called by:	–
Subalgorithm of:	GetEncryptionParameters
General comments:	The integer $s_p = 8530092$ and its binary representation $0b100000100010100010101100$ in the comment are incorrect (the bit at index 19 is set to 0), which implies that in Line 3 the algorithm wrongly returns false for the prime number $n = 19$. The correct value is $s_p = 9054380$ ($0b100010100010100010101100$). Furthermore, we recommend defining s_k as the product of the first k primes $p_1, \ldots, p_k$ (e.g. $s_9 = 203693490$ for the product of all primes up to $p_9 = 23$) and to use $\gcd(n, s_k) = n$ for detecting them in the test. The same value s_k could then be used for detecting corresponding composite numbers larger than p_k by $\gcd(n, s_k) > 1$ (as an extension for the test in Lines 5–7). [updated in November release: algorithm removed]

Code comments:	The available code does not include an implementation of this algorithm. Instead, the implementation of GetEncryptionParameters calls the standard method `BigInteger::isProbablePrime`, which does not correspond to this algorithm. Primality tests are therefore not implemented in alignment with the specification. [updated in November release: algorithm removed]

Algorithm 4.16: PassesMillerRabin

Called by:	–
Subalgorithm of:	IsProbablePrime
General comments:	Since the Miller-Rabin test is standard in cryptography, we do not see the necessity of giving the algorithm in pseudocode. The necessary extension for the Baillie-PSW test could be described in the text. [updated in November release: the algorithms has been removed]
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

Algorithm 4.17: GetLucasSequenceValues

Called by:	–
Subalgorithm of:	IsProbablePrime
General comments:	*none*
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

Algorithm 4.18: GetLucasParameters

Called by:	–
Subalgorithm of:	IsProbablePrime
General comments:	*none*
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

Algorithm 4.19: LucasDoubleK

Called by:	–
Subalgorithm of:	GetLucasSequenceValues
General comments:	The name of this algorithms is in conflict with the general algorithm naming conventions. [updated in November release: algorithm removed]
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

Algorithm 4.20: LucasIncrementK

Called by:	–
Subalgorithm of:	GetLucasSequenceValues
General comments:	The name of this algorithms is in conflict with the general algorithm naming conventions. [updated in November release: algorithm removed]
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

Algorithm 4.21: IsPerfectSquare

Called by:	–
Subalgorithm of:	GetLucasParameters
General comments:	*none*
Code comments:	The available code does not include an implementation of this algorithm. [updated in November release: algorithm removed]

3.2.4. Symmetric Authenticated Encryption

The two algorithms in this category deal with symmetric encryption. There are a few deviations between code and specification, which we think should be addressed. What is missing from the point of view of the definition of a symmetric encryption scheme is a key generation algorithm, which "knows" the type and length of the keys to generate (possibly as a function of the security parameter). Otherwise, the implementation of the key generation is left to a programmer who may not be aware of the delicacy of this task and the details to consider.

Algorithm 5.1: GenCiphertextSymmetric

Called by:	–
Subalgorithm of:	GenCMTable, GenCredDat

[7] Algorithm RecHash from the current version of [HKLD22a] could be considered as an example for solving the raised issues.

[8] See `https://datatracker.ietf.org/doc/rfc9106`.

[9] We are aware of the current `OWASP` cheat sheet recommending $p = 1$, $m = 15$, and $t = 2$ as a bare minimum for Argon2id, but then again without reasonable justification.

General comments:	Concatenating the byte array representations of the associated data may create unintended "collisions". [updated in October release] The wrong concatenation symbol is used (| instead of ||). [updated in October release] Since $p = \|\mathsf{P}\|$ is not a fixed variable, it would be better to define the plaintext as an element of $\mathcal{B}^*$ (despite the upper limit of p in actual implementations). Same remark for the output ciphertext $\mathsf{C} \in \mathcal{B}^*$. The call of AuthenticatedDecryption uses the wrong font. [updated in December release] The *encryption key* should be called *secret key* or *symmetric key*.
Code comments:	By concatenating `associated_i_bytes.length` to the byte array representations of each associated data, the implementation in `SymmetricAuthenticatedEncryptionService::genCiphertextSymmetric` deviates from the pseudocode. Another deviation is the restriction `stringToByteArray(associated_i).length <= 255` in Line 55, which is not present in the pseudocode. [updated in October release] Furthermore, making a copy of `associatedData` is not necessary. Finally, calling a pair `(C, nonce)` a `SymmetricCiphertext` is somehow misleading, because `C` alone is the ciphertext.

Algorithm 5.2: GetPlaintextSymmetric	
Called by:	–
Subalgorithm of:	GetKey, ExtractCRC, ExtractVCC
General comments:	Same remarks as for GenCiphertextSymmetric.
Code comments:	Same remarks as for GenCiphertextSymmetric.

3.2.5. Digital Signatures

The algorithms in this category deal with signature generation and verification. The main issue here is the question of how to handle the certificate's expiry date. In our comments, we argue that judging the validity of a certificate should only depend on the dates of the given election period, not on the current timestamp. This is a conceptual question, which should be carefully addressed somewhere in the specification document. Such a discussion is currently missing.

Algorithm 6.1: GenKeysAndCert	
Called by:	–
Subalgorithm of:	–
General comments:	This algorithm is never called in the protocol. We recommend making corresponding calls more explicit. Using set operations for defining the value assigned to the variable *info* is slightly abusive.

Code comments:	The name of the helper class `KeysAndCert` (*keys* in plural) is misleading, since it contains only a single private key.
Algorithm 6.2: GenSignature	
Called by:	Called by the authorities for every message sent during a protocol execution.
Subalgorithm of:	–
General comments:	a) While RecursiveHash is called with multiple parameters at every other invocation in the protocol, GenSignature passes a pair (m, c) to RecursiveHash instead of two values m and c. The reason for this is unclear. For consistency reasons, we recommend changing it accordingly. [updated in December release] b) The algorithm Sign(K, M) from the RSASSA-PSS standard expects the message M to be encoded as an octet string. Since the hash value h obtained from calling RecursiveHash is not an encoding of the values to be signed in the strict sense, the corresponding call Sign(privKey, h) is slightly abusive. We recommend adding some clarifying comments to the specification. c) Checking the validity period of the certificate against the current timestamp does not make much sense here (checking the certificate expiry date must be part of the signature verification, see below).
Code comments:	*none*
Algorithm 6.3: VerifySignature	
Called by:	Called by every party receiving a signed message from an authority during a protocol execution.
Subalgorithm of:	VerifyConfigPhase, VerifyTally
General comments:	The above general comments a) and b) for GenSignature apply here for the same reasons. [updated in December release: only for a)] Additionally, we believe that the validity of the certificate should be checked against the end of the election process (to ensure that the certificate was valid at the time of the election), not against the current timestamp. Otherwise, if valid signatures suddenly turn invalid when a certificate expires, the verification of an election is no longer deterministic for a given set of election data.
Code comments:	*none*

3.2.6. ElGamal Cryptosystem

This category of algorithms deals with the ElGamal encryption scheme. It includes some basic algorithms for group parameters and key pair generation, encryption and decryption, and for exploiting the scheme's homomorphic property. The implementation deals with an optional extension called *multi-recipient ElGamal*, in which the same message

can be encrypted for different public keys using a single randomization. This extension, however, is currently not used in the protocol (it may be used in future versions for handling election with write-ins). In the current protocol version, the implementation of this extension makes both the pseudocode and the Java code unnecessarily complicated for no real benefit, hence we recommend replacing it by a standard ElGamal implementation. In the current implementation, for consistency reasons, we recommend renaming the classes `ElGamal`, `ElGamalFactory`, and `ElGamalService` into `MultiRecipientElGamal`, `MultiRecipientElGamalFactory`, and `MultiRecipientElGamalService`, respectively (or removing the term "`MultiRecipient`" from all ElGamal classes).

With respect to the current design of the ElGamal implementation, the introduction of data classes for private and public keys, key pairs, messages, and ciphertexts is possibly not the best decision. In each case, it creates an additional abstraction layer, which makes the handling of corresponding objects more cumbersome (numerous wrapping and unwrapping operations are required at many locations in the code), without providing much benefit. From a theoretical point of view, a multi-recipient ElGamal message, for example, is simply a vector of elements of $\mathbb{G}_q$, and this is how they are characterized in the pseudocode algorithms. The motivation and benefit of introducing a class `ElGamalMultiRecipientMessage` on top of the classes `GqElement` and `GroupVector` is therefore not obvious. Generally, we believe that the potential for simplifications is quite large in this area of the code.

Algorithm 7.1: GetEncryptionParameters	
Called by:	–
Subalgorithm of:	VerifyEncryptionParameters
General comments:	(1) This algorithm is never called in the protocol, except for verifying its output values by the verifier. We recommend making the calls during the configuration phase more explicit by stating when and by whom the algorithm is called. (2) On Line 5, it is confusing to first convert q_b into an integer and then apply a bit shift, i.e., we simply recommend dividing the integer by 4. Furthermore, we recommend swaping Line 8 and Line 9 to avoid repeating the calculation of $2q + 1$. [updated in December release: only the swaping of Lines 8 and 9] (3) Since 3 is also a generator for every subgroup $\mathbb{G}_q$, similar to 4, it is possible to replace the code from Lines 10–14 by a single if-then-else statement (the option $g = 4$ can be dropped). This follows from the fact that every safe prime $p > 7$ satisfies $p \equiv 11 \pmod{12}$. Note that this condition can be used for speeding up the generation of safe primes [Wie03]. [updated in December release]
Code comments:	The method `ElGamalService::getEncryptionParameters` explicitly performs primality tests on p and q, thus its is not necessary to perform the same tests again in the constructor `GqGroup::new`.

Algorithm 7.2: GetSmallPrimeGroupMembers	
Called by:	–
Subalgorithm of:	VerifySmallPrimeGroupMembers
General comments:	(1) This algorithm is never called in the protocol, except for verifying its output values by the verifier. We recommend making the calls during the configuration phase more explicit by stating when and by whom the algorithm is called. (2) The reasons for excluding $g \in \{2, 3, 4\}$ from the search and for imposing upper limits on r remain unclear (the algorithm correctly returns $\perp$ if r is bigger than the number of available primes in $\mathbb{G}_q$).
Code comments:	Limiting the possible primes to integers (of type `int`) smaller than `Integer.MAX_VALUE` is a deviation from the pseudocode. Similarly, testing `!current.equals(gqGroup.getGenerator().value)`, which is not even necessary for $g \in \{2, 3, 4\}$ and `current >= 5`, is not included in the pseudocode. [updated in December release]

Algorithm 7.3: IsSmallPrime	
Called by:	–
Subalgorithm of:	GetSmallPrimeGroupMembers
General comments:	Since the algorithm is a general (even though quite inefficient) solution for testing primes of any size, we recommend calling it IsPrime (no test for "smallness" is included). Furthermore, we recommend improving it according to the "Optimized School Method", which exploits $p \equiv \pm 1 \pmod 6$ for all primes $p > 3$. [updated in December release]
Code comments:	In the implementation, the statement `return true;` in Line 49 should be within the brackets of the `else`-statement (between Line 46 and 47), like in the pseudocode. [updated in December release]

Algorithm 7.4: GenKeyPair	
Called by:	–
Subalgorithm of:	GenKeysAndCert, GenKeysCCR, GenVerDat, SetupTallyCCM
General comments:	The statement that ElGamal encryption is secure for $(sk, pk) = (0, 1)$ is clearly wrong. In that particular case, where pk is not a generator of $\mathbb{G}_q$, we always get $\mathsf{Enc}_{pk}(m, e) = (g^r, m)$. The adversary can then define a simple algorithm to win the ciphertext indistinguishability game with probability 1 for arbitrary pairs of messages. From a theoretical point of view, it is therefore crucial to exclude $pk = 1$ from the key generation algorithm, even if this case will almost never occur for randomly selected private keys.

Code comments:	Using stream programming for getting the random private keys and calling a sub-routine `derivePublicKey` for getting corresponding public keys, the matching between the implementation and the pseudocode is not quite obvious. [updated in November release: method `derivePublicKey` still exists, but is not used anymore]

Algorithm 7.5: GetCiphertext

Called by:	–
Subalgorithm of:	GetCiphertextVectorExponentiation, GenShuffle, GetMultiExponentiationArgument, VerifyMultiExponentiationArgument, GetDiagonalProducts, GenVerDat, CreateVote, GetMixnetInitialCiphertexts
General comments:	*none*
Code comments:	Implemented using (possibly parallel) stream programming.

Algorithm 7.6: GetCiphertextExponentiation

Called by:	–
Subalgorithm of:	GetCiphertextVectorExponentiation, GenEncLongCodeShares, CreateVote
General comments:	By assigning the newly calculated values to the same variables, for example $\phi_i \leftarrow \phi_i^a \bmod p$, it is unclear whether the algorithm mutates the original input vector C_a or computes a new one from scratch. The limiting case $\ell = 0$ is allowed here, but not in GetCiphertext. We recommend adding the constraint $\ell > 1$ to the pseudocode. [updated in December release: only $\ell > 1$ has been added]
Code comments:	Implemented using (possibly parallel) stream programming. We recommend implementing the distinction between sequential and parallel processing in the same way as in GetCiphertext or in GetCiphertextVectorExponentiation.

Algorithm 7.7: GetCiphertextVectorExponentiation

Called by:	–
Subalgorithm of:	GetShuffleArgument, VerifyShuffleArgument, VerifyMultiExponentiationArgument, GetDiagonalProducts
General comments:	The limiting case $\ell = 0$ is allowed here, but not in GetCiphertext. We recommend adding the constraint $\ell > 1$ to the pseudocode. [updated in December release]
Code comments:	Implemented using (possibly parallel) stream programming. The implementation excludes an input of size $n = 0$, but the peudocode allows it. If $n = 0$ is excluded, then the reduction to the identity ciphertext is not necessary.

Algorithm 7.8: GetCiphertextProduct

Called by:	–

Subalgorithm of:	GetCiphertextVectorExponentiation, GetShuffleArgument, VerifyMultiExponentiationArgument, GetDiagonalProducts, CombineEncLongCodeShares
General comments:	The limiting case $\ell = 0$ is allowed here, but not in GetCiphertext. We recommend adding the constraint $\ell > 1$ to the pseudocode. [updated in December release]
Code comments:	Implemented using (possibly parallel) stream programming.

Algorithm 7.9: GetMessage

Called by:	–
Subalgorithm of:	GetPartialDecryption, CombineEncLongCodeShares, GenCMTable
General comments:	*none*
Code comments:	Implemented using (possibly parallel) stream programming.

Algorithm 7.10: GetPartialDecryption

Called by:	–
Subalgorithm of:	GenVerifiableDecryptions
General comments:	*none*
Code comments:	*none*

Algorithm 7.11: GenVerifiableDecryptions

Called by:	–
Subalgorithm of:	MixDecOnline, MixDecOffline
General comments:	*none* [updated in December release: unnecessary restriction $N > 0$ introduced]
Code comments:	Implemented using stream programming. Instead of a single loop as in the pseudocode, stream programming requires two independent statements. Furthermore, the implementation contains a test `C.isEmpty()` to exclude the special case $N = 0$ for no obvious reason. This restriction is not aligned with the pseudocode. [updated in December release: restriction $N > 0$ added to specification, instead of removing it from code]

Algorithm 7.12: VerifyDecryptions

Called by:	–
Subalgorithm of:	VerifyMixDecOnline, VerifyMixDecOffline, VerifyTallyControlComponentBallotBox
General comments:	There is no obvious reason for the restriction $N > 0$, which is not present in GenVerifiableDecryptions. This means that the decryption of an empty list of ciphertexts can not be verified. [updated in December release: restriction $N > 0$ added to GenVerifiableDecryptions] We generally recommend including limiting cases as much as possible.

Code comments:	Implemented using stream programming. Evaluating the conjunction using `Stream::reduce` and an auxiliary class `Verifiable` seems unnecessarily complicated. We recommend using the standard stream method `Stream::allMatch` instead. Furthermore, the restriction $N > 0$ is tested by `1 <= N` instead of `N > 0`. This is obviously the same, but a minor misalignment with the pseudocode. Finally, the test $\gamma \neq \gamma'$ is missing in the implementation. We consider this a major deviation from the specification. [updated in November release: test changed to `N > 0` and additional test for $\gamma \neq \gamma'$ introduced]

Algorithm 7.13: CombinePublicKeys

Called by:	–
Subalgorithm of:	GenVerCardSetKeys, SetupTallyEB, VerifyMixDecOnline, MixDecOnline, VerifyMixDecOffline
General comments:	Either the domain $\mathbb{G}_q^{N \times s}$ of the given matrix of public keys or the indexing in $\mathsf{pk}_{j,i}$ is wrong, since i iterates over N and j over s, not vice versa. Furthermore, the restriction $N > 1$ is missing for being aligned with the restriction $N \in \mathbb{N}^+$ in GenKeyPair. [updated in December release: the indexing problem still exists]
Code comments:	Implemented using stream programming.

3.2.7. Mix Net

The Bayer-Groth mix-net implementation is a central part of the crypto-primitives component. It provides two top-level algorithms GenVerifiableShuffle and VerifyShuffle, i.e., all other algorithms in this section are sub-routines to be called as part of the shuffling or verification procedure. Many of these sub-routines are relatively complex, but this only reflects the general complexity of the Bayer-Groth method. We have two general remarks.

First, there is a lack of generality with respect to the allowed input size N, which cannot be smaller or equal to 2. Since the Bayer-Groth method uses a $n \times m$-matrix of size $N = nm$, we see that the limiting case $N = 0$ may possibly be excluded inherently, but the method should at least be capable of handling $N = n = m = 1$ correctly. We have not analyzed this case in detail, but it is generally a bad sign for either the specification or the implementation, if limiting cases are not included automatically. We highly recommend enhancing the method to include the general case $N \geqslant 0$, because otherwise these cases must always be handled separately in each application, like for example in GetMixnetInitialCiphertexts, where two dummy ciphertexts are added to the list of ciphertexts to achieve $N \geqslant 2$ artificially (only to remove them after performing the mixing and decryption). This is clearly a very unsatisfactory solution.

Second, there is no obvious justification for selecting m and n closest to $\sqrt{N}$, because this only optimizes the size of the resulting proof (not the performance), and the benefit strongly depends on the factorization of N (for example if N is prime, the proof size can not be optimized). We have already discussed this point in [HKLD22c, Section 2.3], but our recommendation has not been considered. Besides optimizing the performance, which we think should be prioritized, our recommendation for $n = N$ and $m = 1$ has a huge potential for simplifications, because then it would be possible to eliminate the algorithms GetProductArgument, VerifyProductArgument GetHadamardArgument, VerifyHadamardArgument, GetZeroArgument, VerifyZeroArgument, ComputeDVector, and GetMatrixDimension, both in the specification document and in the code. This would make the Bayer-Groth shuffle proof implementation much more accessible (the total number of code line for the shuffle proof could be decreased by approximately 30% from currently 5'747 to 3'592). Given the complexity of these algorithms and their implementation, simplifications of such dimensions should be taken into account, especially if their purpose and benefit is more than questionable.

Algorithm 8.1: GenVerifiableShuffle	
Called by:	–
Subalgorithm of:	MixDecOnline, MixDecOnline
General comments:	The restriction $2 \leqslant N$ seems arbitrary. Clearly, shuffling a list of size $N = 0$ or $N = 1$ are trivial cases, but they may well appear in applications. Excluding them limits the generality of this algorithm for no obvious benefit. Furthermore, the second restriction $N \leqslant q - 3$ has no practical relevance.
Code comments:	Using a helper method `CommitmentKeyService::canGenerateKey` for checking $N \leqslant q - 3$ is unnecessarily complicated, especially since `q` is computed explicitly in the following line.
Algorithm 8.2: VerifyShuffle	
Called by:	–
Subalgorithm of:	VerifyMixDecOnline, VerifyMixDecOffline, VerifyTallyControlComponentBallotBox
General comments:	Same remarks about the restriction $2 \leqslant N \leqslant q-3$ as for GenVerifiableShuffle.
Code comments:	Same remark as for GenVerifiableShuffle.
Algorithm 8.3: GenShuffle	
Called by:	–
Subalgorithm of:	GenVerifiableShuffle
General comments:	*none*

Code comments:	Implemented using stream programming. The code line `.flatMap(i -> Stream.of(i)...)` is very confusing and at the same time not necessary (the three `map` operations could be merged into a single `mapToObj` operation, which is directly applied to the stream source `IntStream.range(0, N)`). [updated in December release]

Algorithm 8.4: GenPermutation

Called by:	–
Subalgorithm of:	GenPermutation
General comments:	The restriction $N \in \mathbb{N}^+$ is not necessary. The post-condition $\mathbb{Z}_{N-1}^N$ is incorrect (it should be $\mathbb{Z}_N^N$) and unnecessary (it is satisfied automatically). [updated in December release: only $\mathbb{Z}_{N-1}^N$ has been change to $\mathbb{Z}_N^N$]
Code comments:	*none*

Algorithm 8.5: GetMatrixDimensions

Called by:	–
Subalgorithm of:	GenVerifiableShuffle, VerifyShuffle
General comments:	The special case $N = 1$ should not be excluded, since the algorithm would simply return $m = n = 1$, which is the correct result in that particular case.
Code comments:	Defining `floorSquareRoot` and then calling it at Line 43 does not contribute to better code readability. [updated in December release]

Algorithm 8.6: GetVerifiableCommitmentKey

Called by:	–
Subalgorithm of:	GenVerifiableShuffle, VerifyShuffle
General comments:	Adding g_i to v should be denoted by $v \cup \{g_i\}$. [updated in December release] Assuming that the restriction $\nu \leqslant q-3$ is due to the constraint $w \notin \{0, 1, g\}$, it should be $\nu \leqslant q-2$, because 0 is not an element of $\mathbb{G}_q$.
Code comments:	*none*

Algorithm 8.7: GetCommitment

Called by:	–
Subalgorithm of:	GetCommitmentMatrix, GetCommitmentVector, GetMultiExponentiationArgument, VerifyMultiExponentiationArgument, GetProductArgument, GetHadamardArgument, VerifyHadamardArgument, GetZeroArgument, VerifyZeroArgument, GetSingleValueProductArgument, VerifySingleValueProductArgument
General comments:	*none*

Code comments:	The code line `nu = ck.size()` is incorrect, it should be `nu = ck.size() - 1`. The code for testing the constraint $l > 0$ is missing. If such a test is added to the code, then the test `a.isEmpty()` in Line 73 can be dropped. [updated in November release] A quite complicated single stream programming code line is used to essentially compute a product of powers. We recommend introducing an additional abstraction for this particular operation (it could then also be used to simplify GetCiphertextVectorExponentiation, which performs essentially the same operation in a different context).

Algorithm 8.8: GetCommitmentMatrix

Called by:	–
Subalgorithm of:	GetCommitmentVector, GetShuffleArgument, VerifyShuffleArgument, GetMultiExponentiationArgument, GetProductArgument, GetHadamardArgument, GetZeroArgument
General comments:	The constraint $m, n > 0$ is correct to exclude an empty matrix, but since it allows the special case $m = n = 1$, it is in conflict with the restriction $2 \leqslant N = nm$ from GenVerifiableShuffle.
Code comments:	Implemented using stream programming. Like in GetCommitment, the code line `nu = ck.size()` is incorrect, it should be `nu = ck.size() - 1`. A test for checking the constraint $m, n > 0$ is missing. [updated in November release: comment added]

Algorithm 8.9: GetCommitmentVector

Called by:	–
Subalgorithm of:	GetZeroArgument
General comments:	Since $\mathbf{d}$ is a vector and not a matrix, passing it as a $(2m + 1) \times 1$ matrix to GetCommitmentMatrix is slightly abusive notation. Furthermore, in its current form, the two-lines algorithm could be reduced to a single-line algorithm, which simply returns the result of calling `GetCommitmentMatrix`. But this raises the question, if this algorithm is really necessary, since it does nothing more than checking that both $\mathbf{d}$ and $\mathbf{t}$ have an odd number $2m + 1$ of elements. What is missing in the pseudocode is the transformation of $\mathbf{d}$ into a matrix.
Code comments:	The definition of the variable `d_matrix` in Line 165 implements the missing line in the pseudocode. A check for testing that the length of the input vectors is an odd number $2m + 1$ is missing. [updated in November release]

Algorithm 8.10: StarMap

Called by:	–
Subalgorithm of:	GetZeroArgument, VerifyZeroArgument, ComputeDVector

General comments:	This algorithm is called three times using the symbol $\star$ instead of its name StarMap. We recommend using its name for reasons of consistency. Since the algorithm essentially evaluates a polynomial, one might consider using Horner's algorithm for speeding up the computation.
Code comments:	Implemented using stream programming. The case $n = 0$ is correctly included in the algorithm as a special case. Treating it explicitly as a special case by testing `a.isEmpty()` is not necessary.

Algorithm 8.11: GetShuffleArgument

Called by:	–
Subalgorithm of:	GenVerifiableShuffle
General comments:	*none*
Code comments:	The verification of $n \leqslant \nu$ is done implicitly by verifying against length of $\boldsymbol{ck}$. But this results in $n \leqslant (\nu + 1)$.

Algorithm 8.12: VerifyShuffleArgument

Called by:	–
Subalgorithm of:	VerifyShuffle
General comments:	*none*
Code comments:	See check for ν in GetShuffleArgument.

Algorithm 8.13: ToMatrix

Called by:	–
Subalgorithm of:	GetShuffleArgument, VerifyShuffleArgument
General comments:	*none*
Code comments:	Implemented using stream programming.

Algorithm 8.14: Transpose

Called by:	–
Subalgorithm of:	GetShuffleArgument, VerifyShuffleArgument
General comments:	*none*
Code comments:	Implemented using stream programming.

Algorithm 8.15: GetMultiExponentiationArgument

Called by:	–
Subalgorithm of:	GetShuffleArgument

General comments:	Inconsistent usage of description for vectors: $\boldsymbol{x}$ vs. $\vec{x}$. Omit explications in the specification. Here, the explanation comment at Line 8 confuses an implementer: Ensuring $C_{B_m} = \mathsf{GetCommitment}(0, 0, \boldsymbol{ck})$ and $\texttt{GetCiphertext}(\vec{g}^{b_m}, \tau_m, \boldsymbol{ck}) = \texttt{GetCiphertext}(\vec{1}, p, \boldsymbol{pk})$? Does the implementer have to ensure that? (No, it just states a fact)
Code comments:	Implemented using stream programming.

Algorithm 8.16: VerifyMultiExponentiationArgument

Called by:	–
Subalgorithm of:	VerifyShuffleArgument
General comments:	Inconsistent nomenclature for vectors: $\vec{c}_A$ vs. $\boldsymbol{c}_B$
Code comments:	Implemented using stream programming. Suspicious claim by the implementer without any assertion: "`Hash value is guaranteed to be smaller than q`". The claim is implicitly true, but explicitly unknown here. Only when following the caller's hierarchy up one finds its explicit check at MixnetService. We suggest to reference it. [updated in December release: comment clarified]

Algorithm 8.17: GetDiagonalProducts

Called by:	–
Subalgorithm of:	GetMultiExponentiationArgument
General comments:	The usual constraint $m, n \geqslant 1$ for excluding empty input matrices is missing here.
Code comments:	The computation of d_k is not implemented using an explicit call of GetCiphertext. This is not aligned with the pseudocode.

Algorithm 8.18: GetProductArgument

Called by:	–
Subalgorithm of:	GetShuffleArgument
General comments:	*none*
Code comments:	Implemented using stream programming. The requirement $n \leqslant \nu$ is not checked explicitly. However, the implicit check results in $n \leqslant (\nu + 1)$. See check for ν in GetShuffleArgument. [updated in December release]

Algorithm 8.19: VerifyProductArgument

Called by:	–
Subalgorithm of:	VerifyShuffleArgument
General comments:	No requirements for lower bound set for n. [updated in December release]

Code comments:	The requirement $n \leqslant \nu$ is not checked. [updated in December release]

Algorithm 8.20: GetHadamardArgument

Called by:	–
Subalgorithm of:	GetProductArgument
General comments:	*none*
Code comments:	The value ν is set to be the length of $\boldsymbol{ck}$. But the specification states that length of $\boldsymbol{ck}$ is $(\nu + 1)$. Stream programming is used instead of for loops. In the specification, some sums and products (e.g. c_D and t) count from 1, but implementation starts at 0 and therefore needs additional code to compensate for it, which is error prone. We recommend following the specification as precisely as possible.

Algorithm 8.21: VerifyHadamardArgument

Called by:	–
Subalgorithm of:	VerifyProductArgument
General comments:	*none*
Code comments:	The values p, q are taken from input, however, specification requires it to be from context. Requirement of $n > 0$ is not checked explicitly.

Algorithm 8.22: GetZeroArgument

Called by:	–
Subalgorithm of:	GetHadamardArgument
General comments:	*none*
Code comments:	The specification requires $n, m > 0$. However, this is never checked explicitly, but fails due to a side-effect if $m = 0$.

Algorithm 8.23: VerifyZeroArgument

Called by:	–
Subalgorithm of:	VerifyHadamardArgument
General comments:	
Code comments:	The $\perp$ statement of `verifCd` is not aligned to the other $\perp$ statements. The statement "`value should be 1`" should be "`value is not 1`".

Algorithm 8.24: ComputeDVector

Called by:	–
Subalgorithm of:	GetZeroArgument

General comments:	The usual constraint $m, n \geqslant 1$ for excluding empty input matrices is also missing here. [updated in December release] Instead of the comment "break from loop and proceed with next k", it would be better to surround Line 8 with an if-statement that tests for $j \leqslant m$.
Code comments:	According to the algorithm, $m = 0$ is allowed, but the length of the inputs A and B can certainly not be 0 (the current implementation computes $m = -1$ in that case, which does not make sense). This test is missing.

Algorithm 8.25: GetSingleValueProductArgument	
Called by:	–
Subalgorithm of:	GetProductArgument
General comments:	*none*
Code comments:	Most of the context is derived from the input element statement and not explicitly passed to the algorithm. Hence, also no cross checks are performed. x is converted in a group element of size q. What if $x > q$?

Algorithm 8.26: VerifySingleValueProductArgument	
Called by:	–
Subalgorithm of:	VerifyProductArgument
General comments:	*none*
Code comments:	Same remarks as for GetSingleValueProductArgument.

3.2.8. Zero-Knowledge Proofs

This category of algorithms implements four different types of zero-knowledge proofs. Since they can be regarded as special cases of a generic preimage proof, they have all a similar general structure. In the implementation of the algorithms, we observed that the return values of the verification methods are inconsistent: `verifySchnorr`, `verifyExponentiation`, and `verifyPlaintextEquality` return a value of type `boolean`, whereas `verifyDecryption` returns an instance of the class `VerificationResult` (like the method `verifyShuffle` from the mix-net). We recommend making this more consistent.

We also observed that the following code snippet for computing the value $\mathbf{h}_{\mathsf{aux}}$ appears multiple times in almost exactly the same form across the implementations of all proof generation and verification algorithms (only the domain separation string varies from case to case). For reducing the quantity and improving the quality of the code, we recommend implementing an appropriate additional abstraction, for example in form of a static helper method. Furthermore, we recommend that the distinction between empty and non-empty auxiliary information is either removed or explicitly included in

the pseudocode (using comments is not an appropriate way for defining such important technical details).

```
final HashableList h_aux;
if (!i_aux.isEmpty()) {
    h_aux = HashableList.of(HashableString.from(GEN_SCHNORR_PROOF_SERVICE),
            HashableList.from(i_aux.stream()
                    .map(HashableString::from)
                    .toList()));
} else {
    h_aux = HashableList.of(HashableString.from(GEN_SCHNORR_PROOF_SERVICE));
}
```

Figure 6: Code snippet from the class `SchnorrProofService`.

Algorithm 9.1: ComputePhiSchnorr	
Called by:	–
Subalgorithm of:	GenSchnorrProof, VerifySchnorr
General comments:	*none*
Code comments:	*none*

Algorithm 9.2: GenSchnorrProof	
Called by:	–
Subalgorithm of:	SetupTallyCCM
General comments:	The comment "if $\mathbf{i}_{\text{aux}}$ is empty, we omit it" should be made explicit in the algorithm. Otherwise, it looks as if the code in Lines 105–112 does not match with the specification.
Code comments:	*none*

Algorithm 9.3: VerifySchnorr	
Called by:	–
Subalgorithm of:	SetupTallyEB
General comments:	The wrong font ($\mathbf{x}$ instead of x) is used for the variable storing the return value of ComputePhiSchnorr. [updated in December release] For better consistency with GenSchnorrProof, we recommend merging Line 5 and 6. [updated in December release] Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.

Code comments:	For better matching with Line 3 of the pseudocode, we recommend merging the code Lines 153–154. [updated in December release]

Algorithm 9.4: ComputePhiDecryption

Called by:	–
Subalgorithm of:	GenDecryptionProof, VerifyDecryption
General comments:	The number of keys is usually limited to $\ell > 0$ (for example in GenKeyPair), but here no such restriction is imposed. We are not sure if this is on purpose or a mistake.
Code comments:	Implemented using stream programming.

Algorithm 9.5: GenDecryptionProof

Called by:	–
Subalgorithm of:	GenVerifiableDecryptions
General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	Computation of values y_i implemented using stream programming. The constraint $\ell > 0$ remains unchecked. Division in $\mathbb{G}_q$ is implemented using `GqElement::multiply` and `GqElement::invert`. For better readability, we recommend introducing an additional method `GqElement::divide`. [updated in December release: $\ell > 0$ remains unchecked]

Algorithm 9.6: VerifyDecryption

Called by:	–
Subalgorithm of:	VerifyDecryptions, VerifyMixDecOnline, VerifyMixDecOffline
General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	The constraint $\ell > 0$ remains unchecked. Computation of values y_i and c'_i implemented using stream programming.

Algorithm 9.7: ComputePhiExponentiation

Called by:	–
Subalgorithm of:	GenExponentiationProof, VerifyExponentiation
General comments:	The constraint of $n \in \mathbb{N}^+$ being positive could possibly removed.
Code comments:	Implemented using stream programming.

Algorithm 9.8: GenExponentiationProof

Called by:	–
Subalgorithm of:	GenEncLongCodeShares, CreateVote, PartialDecryptPCC, CreateLCCShare, CreateLVCCShare

General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	*none*

Algorithm 9.9: VerifyExponentiation

Called by:	–
Subalgorithm of:	VerifyBallotCCR, DecryptPCC, VerifyVotingClientProofs, VerifyEncryptedPCCExponentiationProofsVerificationCardSet, VerifyEncryptedCKExponentiationProofsVerificationCardSet
General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	Computation of values c'_i implemented using stream programming.

Algorithm 9.10: ComputePhiPlaintextEquality

Called by:	–
Subalgorithm of:	GenPlaintextEqualityProof, VerifyPlaintextEquality
General comments:	*none*
Code comments:	Here again, division in $\mathbb{G}_q$ is implemented using `GqElement::multiply` and `GqElement::invert`, but `GqElement::divide` would be more appropriate (same remark as for GenDecryptionProof). [updated in December release]

Algorithm 9.11: GenPlaintextEqualityProof

Called by:	–
Subalgorithm of:	CreateVote
General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	Same remark about implementing the division $\frac{c_1}{c'_1}$. [updated in December release] Computation of the pair $\mathbf{z} = (b_1 + e{\cdot}r, b_2 + e{\cdot}r')$ using the methods `VectorUtils::vectorAddition` and `VectorUtils::vectorScalarMultiplication` is correct, but not very obvious.

Algorithm 9.12: VerifyPlaintextEquality

Called by:	–
Subalgorithm of:	VerifyBallotCCR, VerifyVotingClientProofs
General comments:	Same comment about $\mathbf{i}_{\text{aux}}$ being empty as above.
Code comments:	Same remark about implementing the division $\frac{c_1}{c'_1}$. For better matching with Line 4 in the pseudocode, we recommend merging the code Lines 216–217. [updated in December release]

3.2.9. TypeScript

The client implementation in JavaScript also depends on some of the basic algorithms defined in [CryptPrim]. They are available in a separate Maven project called crypto-primitives-ts, which provides a TypeScript implementation of a subset of the Java algorithms implemented in crypto-primitives. In the light of the numbers given in Table 2, the size of TypeScript implementation is approximately 60% less than the size of the Java implementation (9'284 code lines compared to 22'428). We have analyzed the TypeScript implementation as systematically as the Java implementation. Here is the complete list of crypto-primitives algorithms implemented in TypeScript:

CutToBitLength	RecursiveHash	GetCiphertextExponentiation
ByteArrayToInteger	RecursiveHashToZq	ComputePhiExponentiation
IntegerToByteArray	RecursiveHashOfLength	GenExponentiationProof
StringToByteArray	HashAndSquare	VerifyExponentiation
StringToInteger	Argon2id	ComputePhiPlaintextEquality
IntegerToString	IsProbablePrime	GenPlaintextEqualityProof
RandomBytes	GenCiphertextSymmetric	VerifyPlaintextEquality
GenRandomInteger	GetPlaintextSymmetric	
GenRandomVector	GetCiphertext	

Note that some of the implemented algorithms, for example VerifyExponentiation and VerifyPlaintextEquality, are never called by the voting client (we see that they are used for performing client-side tests, but then they should be part of the test code). Other algorithms, for example all sub-algorithms of IsProbablePrime, are missing. [updated in December release: algorithms removed from specification] Generally, we recommend to reduce crypto-primitives-ts to what is really needed by the client, but to implement these algorithms to the full extent.

The TypeScript implementation crypto-primitives-ts is heavily inspired by the Java implementation crypto-primitives. The impression is that most of the algorithms were implemented by taking the corresponding Java code and adapting it to TypeScript. It is remarkable how many of our comments about the Java implementation apply in the same way to the TypeScript implementation. For example, the implementation of GenRandomInteger delegates the generation of the random integers to Verificatum's `LargeInteger` in the same way it is delegated to the `BigInteger` class in Java, i.e., the code is very different from the pseudocode in both implementations. Other examples are the respective implementations of the algorithm GenCiphertextSymmetric, which restricts the associated data to 255 characters in exactly the same but unspecified way, and of the algorithm HashAndSquare, which uses the same method `fromSquareRoot` in Java and TypeScript to compute modular squaring, but in a way that its correctness is not obvious. But not only the pure algorithms are very similar, other design aspects of the library have also been adopted from Java. An example for this is the questionable design of the ElGamal

implementation (see Subsection 3.2.6), which deals with many data classes and a mixture of static and non-static methods for the algorithms.

Due to these very strong similarities between the two implementations, we do not list and comment every algorithm separately. Instead of repeating ourselves, we recommend revising the TypeScript implementation right along with the Java implementation. Some issues that are specific to the TypeScript implementation are discussed below:

Primality Testing

Unlike the Java implementation, which delegates primality testing to the Java class `BigInteger`, the TypeScript implementation provides its own primality test. The implemented primality test is an enhanced Baillie-PSW test as specified in [CryptPrim, Section 4.6]. However, the provided TypeScript implementation is not aligned with the specification.

The algorithm isProbablePrime from [CryptPrim] is implemented as part of the internal class `VerificatumBigInteger`, but already the method signature is quite confusing. In the specification, the algorithm has a single parameter n, which denotes the candidate to be tested for primality. In the implementation, however, there is a single parameter `certainty`, which remains completely unused. It is even more confusing that in the exposed `ImmutableBigInteger` class, the certainty parameter is named `n`. Surprisingly, when the primality test is called in the `GqGroup` class, a certainty level is computed based on the bit length of p and passed to the test, although it is ignored later.

In addition to the confusing method signature, the implementation of the primality test does not conform to the specification as well. The deviations are so fundamental that the code cannot be mapped to the specification. Therefore, it can not be properly audited.

Finally, it is a small detail, but it is almost suspicious that like in the Java implementation, 19 is not considered as a prime number in TypeScript either:

```
private static smallPrimes = 1 << 2 | 1 << 3 | 1 << 5 | 1 << 7 | 1 << 11 | 1 << 13 | 1 << 17 | 1 << 23;
```

Immutability

A very positive point from a software design perspective are the three immutable classes `ImmutableBigInteger`, `ImmutableArray`, and `ImmutableUint8Array` (representing an immutable byte array), which extend the toolbox and clearly indicate the classes' immutability property. There are also other classes, such as for example `GqElement` or `GqGroup`, which do not have the term *immutable* in their class name, but which are at least immutable and documented as such. An irritating exception is the TypeScript class `GroupVector`, which is not immutable. Note that the equivalent Java class is immutable and documented as such. Therefore, it could and should be assumed that the TypeScript version is immutable as well, especially if the class is commented with "*This is effectively a decorator for the ImmutableList class*". This inconsistency between the two libraries is very confusing, and it is not

entirely clear whether `GroupVector` is intentionally or unintentionally mutable in TypeScript. This inconsistency should be clarified and fixed.

3.3. System Specification

Similar to the algorithms of the crypto-primitives component [CryptPrim], the 34 algorithms contained in the system specification are well-specified in [SysSpec]. This allows auditors to directly access the associated code within the implementation. Most of the algorithms are implemented in dedicated Java classes, whose class names and method names correspond to the algorithm names according to a clear pattern. This is an important improvement compared to previous versions of the source code and makes it better accessible for reviewers. The general alignment of the source code with the pseudocode is now also at a relatively high and satisfactory level, which makes deviations more recognizable.

Much more complex than the audit of the pure implementation of the algorithms is the question of how the algorithms are embedded in the overall system. This way, it is not always obvious when and by what the algorithms are called and where the context and input values are provided from. In particular, the unsatisfactory and inconsistent treatment of context (see Subsection 3.1) makes traceability tedious. The fact that not all objects are strictly immutable does not simplify the situation. Often, the input object contains `List`s of elements, but the Java interface `List` does not cover immutability. Even if an immutable list is passed when the method is called, this is not apparent from the method signature and can only be determined by a deeper analysis of all the associated code. Improving this situation would be of great benefit not only to the reviewers but also to the robustness and maintainability of the code base.

The algorithms are grouped according to the three general phases of an election: *Configuration Phase*, *Voting Phase*, and *Counting Phase*. A small number of general algorithms are summarized in the section *Preliminaries*.

3.3.1. Preliminaries

The first section of the system algorithms consists of a set of utility algorithms. Although all of these algorithms are important to the protocol, the current specification and implementation give a different impression. For example, the two algorithms EncodeVotingOptions and DecodeVotingOptions are essential to make the protocol verifiable, but the algorithms are neither called in the protocol nor are they implemented. The two algorithms related to `correctnessID` are also not to be found in the system, but reside somewhere in the crypto-primitives-domain component. Given the importance of these algorithms, we believe that the specification and implementation should be revised.

Algorithm 3.1: EncodeVotingOptions

Called by:	–
Subalgorithm of:	–
General comments:	This algorithm is never called in the protocol. We recommend making corresponding calls more explicit.
Code comments:	The code does not include an implementation of this algorithm.

Algorithm 3.2: DecodeVotingOptions	
Called by:	–
Subalgorithm of:	–
General comments:	This algorithm is never called in the protocol. We recommend making corresponding calls more explicit. [updated in October release]
Code comments:	The code does not include an implementation of this algorithm. [updated in October release]

Algorithm 3.3: Factorize	
Called by:	–
Subalgorithm of:	ProcessPlaintexts, VerifyProcessPlaintexts
General comments:	*none*
Code comments:	If the message cannot be factorized, an exception is thrown instead of returning $\perp$ as specified.

Algorithm 3.4: GetCorrectnessIdForSelectionIndex	
Called by:	–
Subalgorithm of:	CreateLCCShare, ExtractCRC
General comments:	*none*
Code comments:	This algorithm is implemented in the *Crypto Primitives Domain*, which is confusing as the algorithm is not specified in the *Cryptographic Primitives Specification*. We recommend moving either the implementation or the specification of the algorithm for a better alignment between specification and implementation.

Algorithm 3.5: GetCorrectnessIdForVotingOptionIndex	
Called by:	–
Subalgorithm of:	GenVerDat, GenCMTable
General comments:	*none*
Code comments:	Same remark as for GetCorrectnessIdForSelectionIndex.

3.3.2. Configuration Phase

Most algorithms of the configuration phase are called by the setup component and are therefore located in the secure-data-manager. Two algorithms are called by the CCR control component and one algorithm by the CCM control component. These three algorithms are located in the control-components. Given the code base, it is not clear why there is still a distinction between CCR and CCM in the specification (see Subsection 3.1). This artificial distinction must be dropped later, during the tallying phase. There, it is important for security reasons that the CCRs and CCMs are indeed the same instances. We suppose that the terms *CCR* and *CCM* are legacy and we suggest to omit the distinction and propose the umbrella term *CC* to eliminate confusion and improve the alignment with the source code.

Algorithm 4.1: GenKeysCCR	
Called by:	CCR
Subalgorithm of:	–
General comments:	*none*
Code comments:	It is not obvious why the random source is passed to the GenKeyPair algorithm explicitly. In most other cases, the random source is part of the underlying service class and not of the algorithm's input. This is confusing and indicates an inconsistent API design. We recommend a consistent handling of the random source throughout the code base.
Algorithm 4.2: GenSetupEncryptionKeys	
Called by:	Setup Component
Subalgorithm of:	–
General comments:	*none*
Code comments:	Same remark as for GenKeysCCR.
Algorithm 4.3: GenVerDat	
Called by:	Setup Component
Subalgorithm of:	–
General comments:	On Line 16, the algorithm GenUniqueDezimalStrings returns a list of strings and not a single value. Please assign the first element of the returned list to $\texttt{BCK}_{\texttt{id}}$ to prevent confusion. [updated in December release: comment added]

Code comments:	The unspecified conversion of $\texttt{vc}_{\texttt{id}}$ and $\texttt{SVK}_{\texttt{id}}$ to lower case is problematic. If it is important that these values are in lower case, then we recommend defining the base16 and base32 alphabets in lower case and that the algorithms GenRandomBase16String and GenRandomBase32String return strings of the corresponding alphabet. Instead of using the specified algorithm Base64Encode for the base64 encoding the class `java.util.Base64` is used. Even though the outcome is the same, not using the specified algorithms results in code duplication which is a blunt source of error for future maintenance. [updated in November release] In addition, the general alignment with the specification could be further improved: lines 4 and 5 are swapped, `p_k` and `primesMappingTableEntries` should be named `p_tilde_k` and `pTable_entries`, respectively. [updated in December release] The two lists `K` and `k` are expected as output and not a list of $(\texttt{K}_\texttt{i}, \texttt{k}_\texttt{i})$ tuples.

Algorithm 4.4: GenEncLongCodeShares

Called by:	CCR
Subalgorithm of:	–
General comments:	The input $\mathbf{c}_{\texttt{pCC}}$ and the output $\mathbf{c}_{\texttt{expPCC,j}}$ are elements of $(\mathbb{G}_q^{n+1})^{\mathtt{N_E}}$ and not of $(\mathbb{G}_q \times \mathbb{G}_q^n)^{\mathtt{N_E}}$ and GenExponentiationProof expects a vector for the bases and a vector for the exponents and not a tuple of a single value and a vector each. For example, $(g, \mathsf{c}_{\texttt{pCC,id}})$ should be $(g, \mathsf{c}_{\texttt{pCC,id}_0}, \ldots, \mathsf{c}_{\texttt{pCC,id}_n})$. The updated list $\mathsf{L}_{\texttt{genVC},j}$ is a side-effect of the algorithm.
Code comments:	The input validation is insufficient as the input is not cross checked with the context. [updated in December release] An unspecified list of `vcPublicKeys` is additionally passed as input to the algorithm and later stored in the list $\mathsf{L}_{\texttt{genVC},j}$. It is not clear what these public keys are used for. If they are needed, why are they not specified.

Algorithm 4.5: CombineEncLongCodeShares

Called by:	Setup Component
Subalgorithm of:	–
General comments:	The input $\mathbf{C}_{\texttt{expPCC}}$ is an element of $\left((\mathbb{G}_q^{n+1})^{\mathtt{N_E}}\right)^4$ and not $\left((\mathbb{G}_q \times \mathbb{G}_q^n)^{\mathtt{N_E}}\right)^4$. Also the set definition of the output $\mathbf{c}_{\texttt{pC}}$ is wrong in the same way.
Code comments:	The input validation is insufficient as the input is not cross checked with the context. [updated in October release] Instead of using the specified algorithm Base64Encode, the encoding is implemented using the `java.util.Base64` class (see remarks for GenVerDat). [updated in November release]

Algorithm 4.6: GenCMTable

Called by:	Setup Component
Subalgorithm of:	–

General comments:	GenUniqueDezimalStrings returns a list of strings and not a single value (see remarks for GenVerDat) [updated in December release: comment added] and wrong set definition for $\mathbf{c}_{\mathtt{pC}}$ (see remarks for CombineEncLongCodeShares)
Code comments:	The specified algorithm Base64Encode is not used (see remarks for GenVerDat). [updated in November release] The final ordering of the CMtable is implemented implicitly by creating a new Java `TreeMap` object. Even though a `TreeMap` orders its elements according to the Java documentation, we would prefer the ordering to be implemented explicitly to make the code more transparent and aligned to the specification.

Algorithm 4.7: GenVerCardSetKeys

Called by:	Setup Component
Subalgorithm of:	–
General comments:	*none*
Code comments:	*none*

Algorithm 4.8: GenCredDat

Called by:	Setup Component
Subalgorithm of:	–
General comments:	The algorithm Argon2id expects a byte array as input argument, but $\mathtt{SVK}_{\mathtt{id}}$ is a base32 string. Hence, $\mathtt{SVK}_{\mathtt{id}}$ needs to be converted before it is passed to the algorithm. [updated in October release]
Code comments:	The algorithm Argon2id is named GenArgon2id and returns not only the tag but also the salt. The salt is concatenated to the ciphertext and to the nonce of $\mathtt{VCks}_{\mathtt{id}}$. It is then returned as part of the base64 encoded $\mathtt{VCks}_{\mathtt{id}}$. This is not in alignment with the specification. [updated in October release] The algorithm Base64Encode is not used (see remarks for GenVerDat). [updated in November release]

Algorithm 4.9: SetupTallyCCM

Called by:	CCM
Subalgorithm of:	–
General comments:	*none*
Code comments:	*none*

Algorithm 4.10: SetupTallyEB

Called by:	Setup Component
Subalgorithm of:	–
General comments:	*none*

Code comments:	No context is provided to the algorithm but it is derived from the input. [updated in December release] The alignment with the specification should be further improved. Instead of returning $\perp$ an exception is thrown and the passwords of the electoral board members `PW` are hashed as a list and not as single values as described in the specification. [updated in December release] In addition, the first loop over the control components is not implemented as a loop but as copy/paste of several lines of code. This is not only inconsistent to the specification but also bad practice. [updated in December release] From an operational perspective, wiping passwords after usage is recommended to reduce the time passwords are in plaintext in memory. However, we think the implementation should be improved. We recommended already several times to work strictly with immutable objects. Therefore we recommend introducing special immutable but wipe-able objects. These objects cannot be changed but provide a clear interface to wipe their data after usage.

3.3.3. Voting Phase

The voting phase is the only phase in which not all algorithms are implemented in Java. Some of them are executed by the voting client and, hence, they are implemented in JavaScript. It is also the only phase in which the system is exposed to the public and many voters may interact with the system exactly at the same time or a single voter may try to cast multiple messages at the same time. It is therefore essential that the system can handle multiple messages in parallel and that a proper synchronization is implemented to guarantee that only a single message of a certain type per voter is accepted. We analyzed both aspects and both of them have been carefully taken care of. Especially synchronization has been properly implemented on a database level which is a major improvement compared to previous versions of the system.

As the voting client is still implemented in an old and no longer supported framework called AngularJS we focused mainly on the pure algorithms. [updated in October release] We were told that the client will be replaced by an implementation based on an up-to-date framework before going live. We are surprised that a partially trustworthy component, which is not secured by cryptographic means but operational means, is replaced at the last moment. The algorithms, and that is why we focused on them, will probably remain the same. But introducing a new framework with all its dependencies inevitably comes with new security risks.

Having a look at the communication between voting client and voting system it is conspicuous that the exchanged messages contain too much information. For example, the cast vote consists not only of the specified data but also of a list of `correctnessIds` and a not further explained `credentialId`. Although we do not see a concrete attack based on the additional information, it is a deviation from the specification which raises

questions and confusions. Similar to the algorithms, we recommend strictly following the specification also for the exchanged messages.

Algorithm 5.1: GetKey	
Called by:	Voting Client
Subalgorithm of:	–
General comments:	The four lines $10 - 13$ dealing with splitting $\texttt{VCks}_{\texttt{id,combined}}$ into the ciphertext $\texttt{VCks}_{\texttt{id,ciphertext}}$ and nonce $\texttt{VCks}_{\texttt{id,nonce}}$ renders the algorithm unnecessarily cumbersome. In addition, exactly the same four lines are repeated in two subsequent algorithms. We recommend introducing a dedicated subalgorithm or changing GetPlaintextSymmetric such that the combined value can be passed directly. Argon2id expects a byte array and not a string (see remarks for GenCredDat). [updated in October release]
Code comments:	The alignment with the specification should be improved. Instead of using the specified Base64Encode and Base64Decode algorithms a third party `Buffer` implementation offering base64 conversions is used. [updated in November release] Also the order of statements is not aligned with the specification. [updated in November release] Finally, the implementation deals with an unspecified salt taken from $\texttt{VCks}_{\texttt{id}}$ and passed to Argon2id (which is named `getArgon2id`). [updated in October release]

Algorithm 5.2: CreateVote	
Called by:	Voting Client
Subalgorithm of:	–
General comments:	Since the voter knows only the voting options, but has no clue about the prime number encoding, the algorithm should receive a list of selected voting options as input, not a list of selected encoded voting options.
Code comments:	The context is derived from the input elements and not passed explicitly to the algorithm. Hence, also no cross checks are performed. [updated in November release]

Algorithm 5.3: VerifyBallotCCR	
Called by:	CCR
Subalgorithm of:	–
General comments:	*none*

Code comments:	The values ψ and $\hat{\delta}$ are not taken from the passed context but retrieved from external services. The reason for this is not obvious. We recommend taking all values exclusively from the context and the input object. This renders the algorithms more independent, more concise, and prevents code duplication. [updated in November release]

Algorithm 5.4: PartialDecryptPCC

Called by:	CCR
Subalgorithm of:	–
General comments:	The list $\mathsf{L}_{\texttt{decPCC},j}$ is taken from the context but the list is also modified during the operation, which is very problematic. In our understanding, the context represents the *ground truth*, which is either received from the election authority over an authentic channel or computed based on the received data during setup. For simplicity, certain values may not be received from the election authority but are configured as system-wide parameters. Modifying the list $\mathsf{L}_{\texttt{decPCC},j}$ violates the immutability of the context and produces side-effects. An algorithm is supposed to transform a given input into an output without side-effects. The same input should always result in the same output (of course, apart from the random values generated during the algorithm), i.e. an algorithm should be idempotent. [updated in November release: introduced new section *Stateful Lists and Maps* but algorithm still produces side-effects]
Code comments:	The values ψ and $\hat{\delta}$ are not taken from the passed context (see remarks for VerifyBallotCCR). [updated in November release] Also, the list $\mathsf{L}_{\texttt{decPCC},j}$ is not part of the context but indirectly accessed using a service during the operation.

Algorithm 5.5: DecryptPCC

Called by:	CCR
Subalgorithm of:	–
General comments:	The definition of $\hat{\mathbf{j}}$ is inaccurate. It should be $\hat{\mathbf{j}} = (\hat{j}_1, \hat{j}_2, \hat{j}_3), \hat{j}_k \in \{1, \ldots, 4\} \backslash j$ [updated in December release]
Code comments:	The values ψ and $\hat{\delta}$ are not taken from the passed context (see remarks for VerifyBallotCCR). Also, $\hat{\mathbf{j}}$ is not taken from the context but computed locally. [updated in November release] The indexing using `k` and `index` is very confusing and very difficult to map to the specification. We strongly recommend improving the alignment for example by introducing dedicated zero- and one-based vector classes. Instead of returning $\perp$, an exception is thrown.

Algorithm 5.6: CreateLCCShare

Called by:	CCR

Subalgorithm of:	–
General comments:	The list $\mathsf{L}_{\texttt{sentVotes},j}$ of the context is altered (see remarks for PartialDecryptPCC). And the algorithm GenExponentiationProof expects a vector as input for the bases and exponents and not a tuple (see remarks for GenEncLongCodeShares).
Code comments:	The value ψ is not taken from the passed context (see remarks for VerifyBallotCCR), the algorithm Base64Encode is not used (see remarks for GenVerDat), [updated in November release] and instead of returning $\perp$, an exception is thrown. In addition, the lists $\mathsf{L}_{\texttt{decPCC},j}$ and $\mathsf{L}_{\texttt{sentVotes},j}$ are not part of the context but indirectly accessed using services during operation.

Algorithm 5.7: ExtractCRC

Called by:	Voting Server
Subalgorithm of:	–
General comments:	The variable $\texttt{CC}_{\texttt{id},i}$ is reassigned, which we do not recommend. It renders the algorithm more difficult to read and to understand and it results immediately in deviations. Beside `CC_id_i`, the current implementation introduces a variable `CC_id_i_plaintext`. [updated in December release]
Code comments:	In the comment of the algorithm it is declared that the algorithm may throw a `JsonProcessingException`. However, this is never the case. [updated in December release] On the other hand, an exception is thrown instead of returning $\perp$ and the algorithm Base64Encode is not used (see remarks for GenVerDat). [updated in November release] A further deviation is the return value, which contains not only the list $\mathbf{CC}_{\texttt{id}}$ but also ψ. [updated in December release]

Algorithm 5.8: CreateConfirmMessage

Called by:	Voting Client
Subalgorithm of:	–
General comments:	*none*
Code comments:	The group of $\texttt{k}_{\texttt{id}}$ is not cross checked with the context. [updated in December release]

Algorithm 5.9: CreateLVCCShare

Called by:	CCR
Subalgorithm of:	–
General comments:	The list $\mathsf{L}_{\texttt{confirmationAttempts},j}$ of the context is altered (see remarks for PartialDecryptPCC) and the list $\mathsf{L}_{\texttt{sentVotes},j}$ is neither part of the context nor input.[updated in November release: introduced *Stateful Lists and Maps*]

Code comments:	The input is not cross checked with the context and g is derived from the input instead of taken from the context. [updated in November release] The algorithm Base64Encode is not used (see remarks for GenVerDat). [updated in November release] Instead of verifying the number of attempts with an explicit `if` statement as specified, a helper function `checkArguments` is used. This results in an inappropriate `IllegalArgumentExcpetion` instead of the specified return value $\perp$. In addition, the lists $\mathsf{L}_{\texttt{confirmedVotes},j}$ and $\mathsf{L}_{\texttt{confirmationAttempts},j}$ are not part of the context but indirectly accessed using services.

Algorithm 5.10: VerifyLVCCHash

Called by:	CCR
Subalgorithm of:	–
General comments:	The list $\mathsf{L}_{\texttt{confirmedVoted},j}$ of the context is altered (see remarks for PartialDecryptPCC) and the list $\mathsf{L}_{\texttt{sentVotes},j}$ is neither part of the context nor input. The definition of $\hat{\mathbf{j}}$ is inaccurate (see remarks for DecryptPCC).[updated in November release: introduced *Stateful Lists and Maps*]
Code comments:	The lists $\mathsf{L}_{\texttt{confirmedVotes},j}$ is not part of the context but indirectly accessed using a service. The algorithm Base64Encode is not used (see remarks for GenVerDat).

Algorithm 5.11: ExtractVCC

Called by:	Voting Server
Subalgorithm of:	–
General comments:	The variable $\texttt{VCC}_{\texttt{id}}$ is reassigned (see remarks for ExtractCRC). [updated in December release]
Code comments:	The CMtable is not immutable, which renders input validation worthless. We strongly recommend working exclusively with immutable objects. It is confusing that two different methods are used to create an empty list (once `List.of()` is used and the other time `Collections.emptyList()`). [updated in December release] If there is a reason to use different methods, then please comment it, otherwise always use the same method to reduce complexity and to raise fewer questions. The algorithms Base64Encode and Base64Decode are not used (see remarks for GenVerDat) and an exception is thrown instead of returning $\perp$.

3.3.4. Tally Phase

The tally phase concludes the protocol with a number of algorithms executed by the control components and the offline tally control component. After the control components

mixed and partially decrypted the recorded votes, the offline tally control component verifies the proofs contained in the ballots and the shuffle and decryption proofs from the control components before performing a final mix and decryption. We do not see any added value in verifying all the proofs by the offline tally component. Verifying the proofs only increases complexity and the time used by the offline tally control component before presenting the final result. But under the given trust assumptions (at least one control component is honest) and the fact that the offline tally control component is not independent from the control components (they both use partially the same code base) it does not increase security. Therefore, we recommend to refrain from executing the two verification algorithms by the offline tally control component.

On the other hand, we question the abrupt end of the protocol. The protocol ends with the offline tally control component presenting a list of prime numbers. However, the prime numbers have no meaning and are only a technical detail used to encode and encrypt a vote. They should not be exposed. The mapping from voting options to prime numbers and vice versa must be performed in a verifiable manner and the final result is then a list of voting options. Only this way, it is not possible to completely invert the final result without being recognized. [updated in October release]

Algorithm 6.1: GetMixnetInitialCiphertexts	
Called by:	CCM
Subalgorithm of:	VerifyOnlineControlComponentsBallotBox
General comments:	Inaccurate notation on Line 6: $\mathbf{c}_{\mathsf{init},j} \leftarrow (\mathbf{c}_{\mathsf{init},j}, \mathbf{E}_{\mathtt{trivial}}, \mathbf{E}_{\mathtt{trivial}})$. Given the context, it is clear what is meant. However, we recommend introducing a dedicated notation for vector concatenation to prevent confusion. Context variables `ee` and `bb` not needed in algorithm.
Code comments:	No context is provided to the algorithm and hence, no cross checking is performed. [updated in December release] The map $\mathbf{vcMap_j}$ is not immutable and therefore the demand for strict immutability is violated.

Algorithm 6.2: VerifyMixDecOnline	
Called by:	CCM
Subalgorithm of:	VerifyOnlineControlComponentsBallotBox
General comments:	Inaccurate notation on Line 13: **if** $\mathsf{decryptVerif}_k \wedge \mathsf{shuffleVerif}_k \; \forall \; \{k\}$ **then**. Please use proper notation for the possible values of k [updated in December release]

Code comments:	The input is not cross checked with the context [updated in December release] and the alignment with the specification should be improved. Instead of two variables $\mathbf{c}_{\mathsf{dec},1}$ and $\pi_{\mathsf{dec},1}$ there is only one variable `dec_1` for an object which holds the values internally. This is especially confusing when calling the algorithm VerifyDecryptions with a different number of arguments than specified. Also confusing is the index k which runs from 1 to $j-1$ and not from 2 to j as specified. Starting at 1 conflicts the variable name `dec_1` which then should be named `dec_0`. The final if-statement is missing. Aspects of it have been moved into the preceding for-loop, which is difficult to map to the specification. [updated in December release: but should be further improved]

Algorithm 6.3: MixDecOnline

Called by:	CCM
Subalgorithm of:	–
General comments:	In the description of the context, it is confusing to describe $\mathsf{L}_{\mathtt{bb},j}$ as a "List of shuffled and decrypted ballot boxes", if a list of ballot box IDs is meant. The updated list $\mathsf{L}_{\mathtt{bb},j}$ of such IDs is a side-effect of the algorithm. We recommend removing the requirement $\mathtt{bb} \notin \mathsf{L}_{\mathtt{bb},j}$ and the side-effect from the algorithm.
Code comments:	The input is not cross checked with the context. [updated in December release] The output has not the structure as defined in the specification and wrongly contains the value $\overline{\mathsf{EL}}_{\mathtt{pk}}$. [updated in December release: structure still not aligned]

Algorithm 6.4: VerifyVotingClientProofs

Called by:	TCM
Subalgorithm of:	VerifyOnlineControlComponentsBallotBox
General comments:	Inaccurate notation on Line 13: **if** $\mathsf{VerifExp}_i \wedge \mathsf{VerifEqEnc}_i \; \forall \; \{i\}$ **then**. Please use proper notation for the possible values of i. [updated in December release]
Code comments:	The input is not cross checked with the context. Minor variable name inconsistencies: `pk_CCR_tilde` should be `pk_tilde_CCR` to be consistent with `E1_tilde_1_i` and `E2_tilde_i` and `Phi_1_0` should be `phi_1_0`. [updated in December release: still minor discrepancies]

Algorithm 6.5: VerifyMixDecOffline

Called by:	TCM
Subalgorithm of:	–
General comments:	
Code comments:	The input is not cross checked with the context. [updated in December release]

Algorithm 6.6: MixDecOffline

Called by:	TCM
Subalgorithm of:	–
General comments:	In the description of the context, it is confusing to describe $\mathsf{L}_{\mathtt{bb,Tally}}$ as a "List of shuffled and decrypted ballot boxes", if a list of ballot box IDs is meant. The updated list $\mathsf{L}_{\mathtt{bb,Tally}}$ of such IDs is a side-effect of the algorithm. We recommend removing the requirement $\mathtt{bb} \notin \mathsf{L}_{\mathtt{bb,Tally}}$ and the side-effect from the algorithm.
Code comments:	The context is derived from the input and therefore no cross checking is possible. [updated in December release] The output has not the structure defined in the specification. Regarding the wiping of passwords see remarks for SetupTallyEB. The comment "`EB_pk is computed during the generation of the key pair (EB_pk, EB_sk)`" should not be necessary. It is a sign that the design of the ElGamal algorithms could be further improved for reaching a better alignment with the specification.

Algorithm 6.7: ProcessPlaintexts

Called by:	TCM
Subalgorithm of:	–
General comments:	A list of integer factorizations should not be regarded as the election result. The decoding step is missing (see discussion in Subsection 3.1.3). [updated in October release] The decrypted write-ins $\psi_{i,1}, \ldots, \psi_{i,l-1}$ are ignored. [updated in November release] The context variables g, `ee`, and `bb` are not needed in the algorithm.
Code comments:	The context is derived from the input and therefore no cross checking is possible.[updated in December release]

3.4. Verifier

Given the late release date in mid-August, the start of our analysis of the verifier component has been postponed compared to the other components. Note that an update of the verifier specification [VerSpec] has been released together with the code on August 19. Since this update included some substantial changes such as an additional algorithm, we did not have much time left to analyse both the specification and the code in the same depth as the other components. The discussion included in this subsection summarized the outcome of our analysis.

The verifier specification makes a distinction between seven so-called *verifications* and five algorithms. The verifications are numbered quite inconsistently from 1.01 to 1.03, from 1.21 to 1.22, and from 2.01 to 2.11, whereas the algorithms are numbered from

3.1 to 3.2 and from 4.1 to 4.3. Unfortunately, the difference between verifications and algorithms is not clear, especially since they are all specified in pseudocode.

Implicitly, there are two additional top-level verification algorithms VerifyConfigPhase and VerifyTally for the two different verification phases, but unfortunately their pseudocode is missing for no obvious reason. To improve the clarity and overall structure of the verifier specification, we recommend calling all specified routines *algorithms* (as we will do in the subsequent discussion), introducing a consistent numbering across all algorithms, and providing pseudocode for the two top-level algorithms .[updated in December release] .

The general purpose and motivation of the verifier specification document is described by the following statement from the abstract of [VerSpec]:

> *"Therefore, this document serves as a manual for developing an independent verifier software and validating or extending the Swiss Post verifier."*

However, we believe that the current document does not fulfill this promise, because many aspects of the verifier are not sufficiently well specified, for example with respect to the completeness, authenticity, consistency, and integrity checks in [VerSpec, Sections 3.1–3.4 and 4.1–4.4]. These checks are only specified very vaguely using sentences such as "*Verify that the primes mapping table pTable of all verification card sets are consistent*". More details are provided for the evidence checks in the algorithms of [VerSpec, Sections 3.5–3.6 and 4.5], but there is also plenty of room left for interpretations. An example are the Lines 2–4 of VerifyOnlineControlComponents, which contain vague sentences such as "*Prepare the Context including the election event context*" instead of proper pseudocode. Therefore, it seems impossible for an external party to build a reliable and independent verifier based only on this document.

Another problem of the specification is the lack of a clear and comprehensive catalog of tests to perform, which is convincing to include all necessary test and nothing else. We observed for example that the zero-knowledge proofs $\pi_{\mathtt{EL_{pk,j}}}$ of the CCMs' election public keys $\mathtt{EL_{pk,j}}$ are not checked by the verifier, but no explanation is given of why this is possibly not necessary. We also observed that some tests are redundant, for example those involving the encryption group in [VerSpec, Section 3.3]. Given the lack of such a test catalog in the verifier specification, we were surprised to find it in the README.md file of the verifier-backend project. As shown in Figure 7, the catalog consists of a total of 39 verifications (24 for verifying the setup and 15 for the tally), of which each has an assigned identifier. We recommend moving this table into the verifier specification and enhancing it with formal descriptions of the respective tests to perform.[10]

Our final general remark concerning the verifier specification is about the differentiation between the (human) *auditor* and the *technical aids* in [OEV, Art. 5]. In the verifier

[10] Some resource files, for example resources_en.properties, contain similar lists of verifications in different languages, but the numbers 200–203 are missing for no obvious reason.

Phase	Id	Name of the verification
Setup	100	VerifySetupCompleteness
Setup	200	CheckSignatureEncryptionParameters
Setup	201	CheckSignatureElectionEventContextData
Setup	202	CheckSignatureOnlineCCKeys
Setup	203	CheckSignatureVerificationData
Setup	204	CheckSignatureEncryptedCodeShares
Setup	205	CheckSignatureTallyData
Setup	300	VerifyEncryptionGroupConsistency
Setup	301	VerifyCCrChoiceReturnCodesPublicKeyConsistency
Setup	302	VerifyCCmElectionPublicKeyConsistency
Setup	303	VerifyChoiceReturnCodesPublicKeyConsistency
Setup	304	VerifyElectionPublicKeyConsistency
Setup	305	VerifyPrimesMappingTableConsistency
Setup	306	VerifyElectionEventIdConsistency
Setup	307	VerifyVerificationCardSetIdsConsistency
Setup	308	VerifyVerificationCardIdsConsistency
Setup	309	VerifyVerificationCardSetsConsistency
Setup	310	VerifyChunkConsistency
Setup	311	VerifyNodeIdsConsistency
Setup	500	VerifyEncryptionParameters
Setup	501	VerifySmallPrimeGroupMembers
Setup	502	VerifyVotingOptions
Setup	503	VerifyEncryptedPCCExponentiationProofs
Setup	504	VerifyEncryptedCKExponentiationProofs
Tally	100	VerifyTallyCompleteness
Tally	200	CheckSignatureBallotBox
Tally	201	CheckSignatureOnlineShuffle
Tally	202	CheckSignatureOfflineShuffle
Tally	203	CheckSignatureProcessedVotes
Tally	300	VerifyConfirmedEncryptedVotesConsistency
Tally	301	VerifyCiphertextsConsistency
Tally	302	VerifyPlaintextsConsistency
Tally	303	VerifyBallotBoxesConsistency
Tally	304	VerifyNumberConfirmedEncryptedVotesConsistency
Tally	305	VerifyEncryptionGroupConsistency
Tally	306	VerifyTallyNodeIdsConsistency
Tally	307	VerifyVotingCardIdsConsistency
Tally	500	VerifyOnlineControlComponents
Tally	501	VerifyTallyControlComponent

Figure 7: Catalog of verification tests as defined in the README.md file of the verifier backend.

specification, by referring to both of them as the *verifier*, this differentiation is not considered:

> *"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."* [VerSpec, Section 1.1]

We understand that this differentiation has been given up for reasons of simplicity, but to check the compliance with the OEV regulations, it creates an additional obstacle. For example, we consider it as essential to see exactly what a human auditor needs to check in addition to the checks performed by the machine operated by the human auditor. Currently, a detailed discussion about how to operate the verifier in practice is somewhat missing.

We also observed that in some cases, the origin of the input data is not clear without any doubts. For example the election event context, which is received from the setup component, also comes with a signature. The verification of this signature is implemented using the class `CheckSignatureElectionEventContextData`, but it is not specified in

[VerSpec]. In that respect, an even more fundamental question arises by considering the definition of the setup component, because according to [OEV Annex, 2.9.2], the setup component cannot be regarded as trustworthy for universal verification. We just wanted to mention this point as a potential violation of the OEV trust model, without having a recommendation for coping with it.

The present verifier implementation calls the cryptographic algorithms of the crypto-primitives components as subroutines. This creates a direct dependency between the two components (see `pom.xml`) and ultimately leads to a common code base between the verifier and the voting system itself. This inherent dependency between the e-voting system and the verification software does not allow for the detection of a wrong result due to implementation errors. However, this poses an immediate risk even if the common code base is open source. The likelihood that an implementation error will not be detected increases if the shared library is built on top of other external dependencies with large code bases. Ultimately, this strong coupling thus does not provide truly independent verification. We consider it inefficient to verify a procedure that has already been verified by at least one honest control component using the same cryptographic means. However, for the verification of the final mixing and decryption step, which takes place at the cantonal level, we consider this dependent verifier to be very valuable. Here, false results due to inadvertent mishandling can be uncovered. Therefore, we will only audit the present implementation under this "reduced" aspect.

Generally, the code implementing the verifier component is well structured and clearly arranged. Therefore, methods implementing the given pseudocode algorithms can be located and inspected easily. We have already mentioned the dependency to crypto-primitives, but there is no direct dependency to the e-voting component. Instead, we found some code that has been copied one-to-one from e-voting to verifier, for example the code implementing the algorithms VerifyMixDecOffline and VerifyVotingClientProof. What is problematical here is the implicit dependency that is created by copying code from one component to another, and also the resulting fact that not all dependencies to other components are handled in the same way. The design strategy leading to the current situation should be reconsidered.

3.4.1. Setup Verification

In this section, we summarize our specific findings relative to the *setup verification phase* (or *config phase*). As already noted, the pseudocode for the top-level algorithm VerifyConfigPhase is missing. We include this algorithm on top of our summary without giving it an algorithm number.

Algorithm ???: VerifyConfigPhase	
Called by:	Auditors
Subalgorithm of:	–

General comments:	The pseudocode of this algorithm is missing.
Code comments:	–

Algorithm 1.01: VerifyEncryptionParameters	
Called by:	–
Subalgorithm of:	VerifyConfigPhase
General comments:	Lines 2–6 could be merged into a single statement **Return** $(p = \hat{p}) \wedge (q = \hat{q}) \wedge (g = \hat{g})$.
Code comments:	–

Algorithm 1.02: VerifySmallPrimeGroupMembers	
Called by:	–
Subalgorithm of:	VerifyConfigPhase
General comments:	Lines 2–6 could be merged into a single statement **Return** $\mathbf{p}' = \mathbf{p}$.
Code comments:	Group, is derived from the input instead of context.

Algorithm 1.03: VerifyVotingOptions	
Called by:	–
Subalgorithm of:	VerifyConfigPhase
General comments:	Inconsistent description of verification for `verifA` and `verifB`. We suggest to fusion this algorithm with VerifySmallPrimeGroupMembers due to overlapping of verifications. Direct verification of $\tilde{p}$ by the result of `GetSmallPrimeGroupMembers`. Lines 2–6 could be merged into a single statement verifA $\leftarrow (\mathbf{p}' = \tilde{\mathbf{p}})$, like in Line 7.
Code comments:	In contrast to specification, the implementation does not use elements from the context. Instead all context-values are derived from 'trusted' input and some values from `VotingOptionsConstants` is used. Pre-validations on $\tilde{p}$ not required, as it eventually is directly compared for equivalence with p.

Algorithm 1.21: VerifyEncryptedPCCExponentiationProofs	
Called by:	–
Subalgorithm of:	VerifyConfigPhase
General comments:	Underspecified. A check is specified using variables (i, j) which are out of scope. Lines 6-10 could be merged into a single line **Return** $\bigwedge_{i,j}$ vcsEncryptedPCCVerif$_{j,i}$.

Code comments:	Not mappable to specification. Some special object for 'contextAndInputs' is taken as input. Then VerifyEncryptedCKExponentiationProofs is called requiring many more parameters. One simply cannot follow that code. Instead of using for-loop programming, parallel stream programming is used in a compact form.
Algorithm 1.22: VerifyEncryptedCKExponentiationProofs	
Called by:	–
Subalgorithm of:	VerifyConfigPhase
General comments:	Same remarks as in VerifyEncryptedPCCExponentiationProofs.
Code comments:	Same remarks as in VerifyEncryptedPCCExponentiationProofs.
Algorithm 3.1: VerifyEncryptedPCCExponentiationProofsVerificationCardSet	
Called by:	–
Subalgorithm of:	VerifyEncryptedPCCExponentiationProofs
General comments:	Under-specified. Groups not cross checked with context. Semantically base g and exponent y are not lists but tuples. However, specification regards them as lists. Questionable naming of exponents as y. Even though, the abuse of $\forall$ is mentioned in the specification, it still is an abuse. The accessibility scope of variables not respected.
Code comments:	Instead of using for-loop programming and following the pseudocode of the specification, parallel stream programming is used in a compact form. Difficult to map to specification.
Algorithm 3.2: VerifyEncryptedCKExponentiationProofsVerificationCardSet	
Called by:	–
Subalgorithm of:	VerifyEncryptedCKExponentiationProofs
General comments:	For tuples vs. list and variable scope see VerifyEncryptedPCCExponentiationProofsVerificationCardSet.
Code comments:	Code quality, see VerifyEncryptedPCCExponentiationProofsVerificationCardSet.

3.4.2. Final Verification

The second verification phase called *final verification* is conducted at the end of the voting process. Again, the pseudocode for the top-level algorithm VerifyTally is missing, but we include it in our summary without giving it a number.

Algorithm ???: VerifyTally

Called by:	Auditors
Subalgorithm of:	–
General comments:	The pseudocode of this algorithm is missing.
Code comments:	–

Algorithm 2.01: VerifyOnlineControlComponents	
Called by:	–
Subalgorithm of:	VerifyTally
General comments:	No acceptable non-ambiguous pseudo code.
Code comments:	Why does the implementer know?

Algorithm 2.11: VerifyTallyControlComponent	
Called by:	–
Subalgorithm of:	VerifyTally
General comments:	See VerifyOnlineControlComponents.
Code comments:	See VerifyOnlineControlComponents.

Algorithm 4.1: VerifyOnlineControlComponentsBallotBox	
Called by:	–
Subalgorithm of:	VerifyOnlineControlComponents
General comments:	The pseudo code of the specification starts to erode, as if the publisher went out of time. This way, the implementer is left on its own, possibly not coming to the same conclusion as the voting system.
Code comments:	–

Algorithm 4.2: VerifyTallyControlComponentBallotBox	
Called by:	–
Subalgorithm of:	VerifyTallyControlComponent
General comments:	See VerifyOnlineControlComponentsBallotBox
Code comments:	–

Algorithm 4.3: VerifyProcessPlaintexts	
Called by:	–
Subalgorithm of:	VerifyTallyControlComponentBallotBox
General comments:	Strange description of the requirement for $\hat{N}_c = N_c$
Code comments:	Stream programming instead of for-loop programming.

A. Addendum-1: October Release

Updates of both the specification and the code were announced on October 3 for the components crypto-primitives and e-voting, and on October 6 for the component verifier. In the sequel, we will refer to this new version as the *October release*, and to the version that was available for our analysis in August (see release history in Figure 1) as the *August release*. Given that the first draft of this document has been submitted to the Federal Chancellery on September 15 (and forwarded to Swiss Post a few days later), i.e., only two weeks before releasing the new version on October 1, we do not expect many of our findings listed in Sections 2 and 3 to be addressed already in the October release. Therefore, the purpose of this addendum section is to give an overview of the changes made in the October release and to verify if any of the changes affects the findings listed in the previous sections of this report. As in Section 3, we structure our analysis according to the changes made in each of the three main system components.

A.1. Overview of Changes

Along with the updated software, the specification documents have also been updated in the October release. The numbering of new document versions suggests that only minor changes have been made to [CryptPrim] (updated from 1.0.0 to 1.0.1), but that some major changes have been made to [SysSpec] and [VerSpec] (updated from 1.0.0 respectively 1.0.1 to 1.1.0). Here is the list of the current documents that we considered in our analysis of the October release (note that [ProtSpec] and [ArchDoc] have not been updated):

- [CryptPrim] *Cryptographic Primitives of the Swiss Post Voting System – Pseudocode Specification*, Version 1.0.1, Swiss Post Ltd., October 3, 2022
- [SysSpec] *Swiss Post Voting System – System Specification*, Version 1.1.0, Swiss Post Ltd., October 3, 2022
- [VerSpec] *Swiss Post Voting System – Verifier Specification*, Version 1.1.0, Swiss Post Ltd., October 3, 2022

As in earlier updates, it is inherently difficult to locate the changes made to these documents, since they are given as PDF-files. However, they all contain a revision chart with links to *change logs* on gitlab.com (see remark below on code changes). Unfortunately, some of these links do not point to the right files.

Regarding the software, moving from pre-release versions 0.15.x.x to a first major release 1.0.0.0, the October release suggests that the project has reached an important milestone, indicating that the software has all major features and is considered reliable enough for general release. Note that in the case of the components crypto-primitives and crypto-primitives-domain, several minor updates have been released in August and September, which apparently represent minor intermediate steps leading to the October release. A

complete overview of the latest versions released on gitlab.com are shown in Figure 8 (the latest minor Version 1.0.0.1 and 1.0.0.2 released shortly after releasing Version 1.0.0.0 were only necessary for creating reproducible builds)

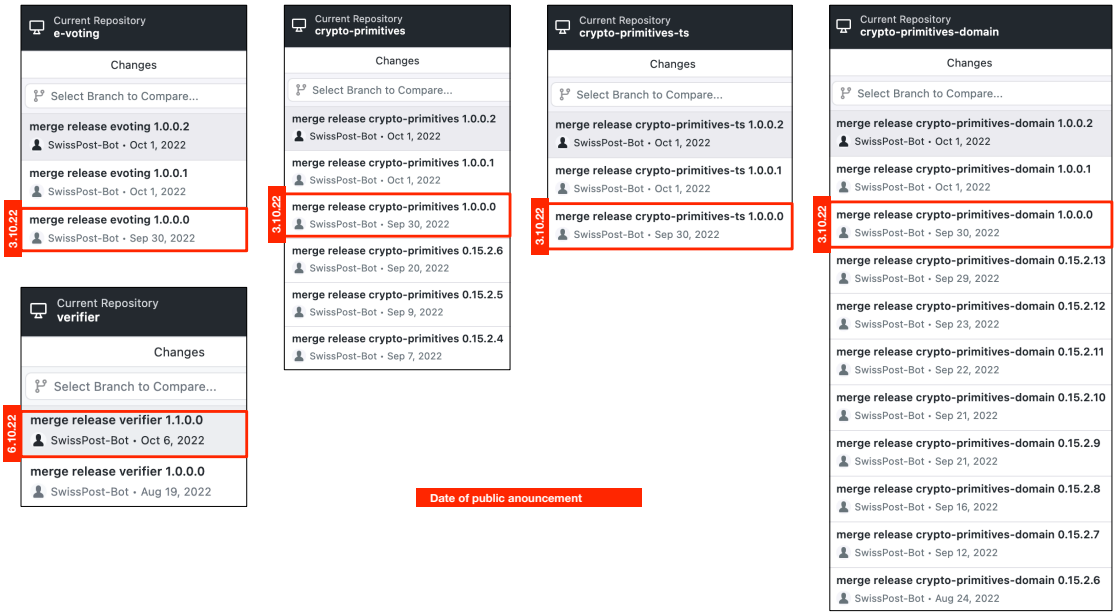


Figure 8: Commit histories of the GitLab projects with dates of public announcements of the October release.

A new feature of the current version is the inclusion of a separate file CHANGELOG.md in all of the five GitLab code repositories for tracking corresponding changes. The information provided in these files turned out to be very useful for getting a first overview of the most relevant changes and as an entry point for our analysis. Some of these files also point out corresponding changes in the specification documents listed above.

The amount of code changes since the August release is reflected by the number of source code files (.java, .js, .ts, .json, ...) appearing in at least one of the GitLab commits between the August and October releases. An overview of these numbers is given in Table 3 (changes in test files are not included). It shows that the largest number of changes has been made to the modules secure-data-manager (backend) and voter-portal of the e-voting component. Many files have changed also in the module domain, but these are mostly auto-generated by JAXB (for example derived from given eCH-xxxx-x-x.xsd files) and are marked as such. Less changes have been made to the crypto-primitives and crypto-primitives-domain components, the modules control-components, tools, and voting-server of the e-voting component, and the modules verifier-backend and verifier-protocol of the verifier component. The awareness of these numbers was an important point of orientation for getting our analysis started at the right places.

Maven Module	.java	.js	.ts	.json	.html	.xsd	.xml	.sql	.properties
crypto-primitives	21	–	–	–	–	–	–	–	–
crypto-primitives-domain	24	–	–	–	–	–	–	–	–
crypto-primitives-ts	–	–	–	2	–	–	–	–	–
command-messaging	3	–	–	–	–	–	–	–	–
control-components	44	–	–	–	–	–	–	1	–
cryptolib	6	–	–	–	–	–	–	–	1
cryptolib-js	–	2	–	3	–	–	–	–	–
domain	227	–	–	–	–	12	1	–	1
secure-data-manager/backend	168	–	–	4	–	–	2	–	1
secure-data-manager/frontend	–	4	–	9	2	–	1	–	–
tools	24	–	–	–	2	–	1	–	1
voter-portal	–	95	205	83	74	–	1	–	–
voting-client-js	–	5	–	2	–	–	1	–	–
voting-server	30	5	–	2	–	–	–	1	2
verifier-assembly	–	–	–	–	–	–	–	–	1
verifier-backend	87	–	–	1	–	–	–	–	–
verifier-frontend	–	–	1	4	1	–	–	–	–
verifier-protocol	14	–	–	–	–	12	1	–	5

Table 3: Number of files changed since the August release (without test files).

A.2. Cryptographic Primitives

The components crypto-primitives, crypto-primitives-ts, and crypto-primitives-domain have only changed moderately since the August version. Most changes correspond to the entries listed in the CHANGELOG.md files and can be located easily in the source files, but there are also some minor changes that are not listed. In the following subsections, we summarize these changes for all three components.

A.2.1. crypto-primitives

There has been several topics with corresponding changes in the code, that have been addressed separately in one of the recent versions. Here is an overview:

- In Version 0.15.1.1, the implementation of the algorithm Argon2id has been split into two methods `genArgon2id` and `getArgon2id` without updating the specification document accordingly. In [CryptPrim, Version 1.0.1], this update has now been delivered. Our second comment on Algorithm 4.14 (Argon2id) is therefore obsolete (all other comments remain valid).

- The recursive hashing method in Algorithm 4.8 has been adjusted twice, first in Version 0.15.2.3 by introducing an additional domain separator prefix <0x03> for lists, and second in Version 0.15.2.5 by removing the limitation to non-empty lists. In [CryptPrim, Version 1.0.1], the description of the algorithm has been updated accordingly. Therefore, the comments in Subsection 3.2.3 about the unnecessary exclusion of empty lists and the missing prefix for vectors become obsolete (all other comments remain valid).
- The processing of the associated data in Algorithm 5.1 (GenCiphertextSymmetric) and of the additional context information in Algorithm 4.12 (KDF) have been updated. To ensure that the encoding becomes injective, additional single bytes representing the length of the associated data or additional information have been introduced. This limits the maximal length to 255 bytes, but this limitation is correctly specified as an additional precondition in [CryptPrim, Version 1.0.1] and implemented accordingly in Version 0.15.2.3 of the code. Our first comments in Subsections 3.2.3 and 3.2.4 about these algorithms are therefore no longer valid (all other comments remain valid).
- In Version 0.15.2.4, parallel stream processing has been introduced at many different places of the code. The unique purpose of this change is to improve the performance of computing large numbers of modular exponentiations. It has therefore no impact on the security and is unrelated to the findings listed in this report.
- The definition of the security levels has been changed in [CryptPrim, Version 1.0.1] and in Version 0.15.2.6 of the implementation. The former security level `DEFAULT` (112 bits, 2028 bits modulus) is now called `LEGACY`, and the security level `EXTENDED` (128 bits, 3072 bits modulus) is now returned as default value by the method `SecurityLevelInternal::getSystemSecurityLevel`. Provided that the resulting decrease of performance has no negative usability impact, particularly on the voting client, we support this change.
- Minor modifications have been made to Algorithm 4.7 (`GenUniqueDecimalStrings`) and Algorithm 7.3 (IsSmallPrime), but our comments in Subsections 3.2.3 and 3.2.6 remain valid.

Note that Version 1.0.0.0 (October release) contains no relevant changes other than the updated pom.xml file and the new specification document.

A.2.2. crypto-primitives-ts

The new Version 1.0.0.0 contains no significant changes compared to the previous Version 0.15.2.3 from July 20, 2022. The only relevant code files with modifications are package.json and package-lock.json (see Table 3), in which nothing but the version number has been updated to 1.0.0.0.

All changes reported in the file CHANGELOG.md were already present in Version 0.15.2.3 from July 20 or earlier. They mainly correspond to the changes reported for the crypto-primitives component: GenArgon2id and GetArgon2id were updated in Version 0.15.1.1 and RecursiveHash in Version 0.15.2.2. By updating the descriptions of these algorithms in [CryptPrim, Version 1.0.1], the reported misalignment between code and specification has been eliminated.

A.2.3. crypto-primitives-domain

Several domain classes have been updated between the August and the October releases. From the total of 24 modified classes (see Table 3), 10 classes were only changed in the comments. The other changes are the following:

- In the classes Ballot, CombinedCorrectnessInformation, Contest, and ElectionAttributes, the changes are restricted to some renamed methods.
- Some domain classes have been changed or extended for various reasons. Some of these changes are listed in the CHANGELOG.md file:
  - ElectionEventContext, ElectionEventContextPayloadDeserializer, ControlComponentPublicKeys: key generation Schnorr proofs added in Version 0.15.2.12
  - TallyComponentShufflePayload: refactored in Version 0.15.2.5
  - SetupComponentVerificationDataPayload: slightly extended in in Version 0.15.2.13
- The most significant change is the introduction of the following domain classes for the extended *primes mapping table* pTable in Version 0.15.2.8:
  - PrimesMappingTable
  - PrimesMappingTableEntry
  - PrimesMappingTableEntryGroupVectorDeserializer

  The reason for this extension is the inclusion of pTable in the exponentiation and plaintext equality proofs generated by the voting client in Algorithm 5.2 (CreateVote) for creating a consistent view across all protocol parties (see remarks in Appendix A.3.2). The new domain classes are needed for transferring this information to the voting client. In VotingOptionsConstants and VerificationCardSetContext, corresponding adjustment had to be made. We have no objections against this extension, except for the fact that pTable is an element of the voting protocol and should therefore not appear in any of the crypto-primitives components.

Generally, the relevance of the above changes in the domain classes is very moderate for our security analysis.

A.3. E-Voting

In the light of the numbers shown in Table 3, there are several areas of major code modifications. We have already noted that the 227 modified Java files in the module domain are auto-generated by JAXB, i.e., they are not relevant for our analysis. Otherwise, the modules with the largest numbers of modified files are control-components, secure-data-manager/backend, voter-portal, and voting-server. Some of the changes are listed in the component's CHANGELOG.md file, but the information given there is not very detailed. For obtaining a more accurate overview of the changes, we considered the GitLab commit history of the developer branch, which includes 150 commit descriptions since Version 0.15.3.0 from July 27, 2022. Among the code areas in which we observed major changes, we consider two of them as relevant from a security perspective. They will be discussed in Appendices A.3.2 and A.3.3. An overview of other changes is given in the following subsection.

A.3.1. Overview of Minor Changes

Many of the entries of the CHANGELOG.md file are minor changes, which affect only a few source files from the e-voting component. Below we discuss the results from analyzing these changes.

- The reported increase of the default security level from 112 bits (2048 bits modulus) to 128 bits (3072 bits modulus) is something that has clearly an impact on the whole system, but it affects only the code of the crypto-primitives component (see remarks in Appendix A.2.1). The entry in the CHANGELOG.md file of the e-voting component is therefore a bit misplaced.
- A small misalignment between code and specification has been corrected on Line 8 of Algorithm 4.8 (GenCredDat). This renders our first comment in Subsection 3.3.2 obsolete. The second entry about GenCredDat in CHANGELOG.md is misleading, because it only announces the introduction of the memory-hard key derivation function Argon2id, but this change was already present in the August release. On the other hand, the important correction in the new release is not mentioned.
- A new algorithm GetElectionEventEncryptionParameters has been added to [SysSpec, Version 1.1.0], but we were unable to locate an implementation of this algorithm in the code (the method `EncryptionParametersAndPrimesGenerator::generate` is very similar, but it includes the generation of a signature). Also, by the information given in the specification, it is not entirely clear when and by which party this algorithm is called. The specification states that the "*setup component generates the election event encryption parameters with the [...] algorithm*", but the algorithm is not called in any of the protocol diagrams (the same remark holds for the two sub-algorithms GetEncryptionParameters and GetSmallPrimeGroupMembers, see remarks in Subsection 3.2.6). We also question the general usefulness of this algorithm,

because it is simply a composition of two existing algorithms, which could also be called separately.[11]

- An implementation of Algorithm 3.2 (DecodeVotingOptions) has been added to the code, and a call to this algorithm has been added to Algorithm 6.7 (ProcessPlaintexts). This addresses our comments in Subsections 3.3.1 and 3.3.4. In the current version, ProcessPlaintexts returns both the list of factorized prime numbers L_{votes} and the list a decoded voting options $L_{\text{decodedVotes}}$. According to the protocol diagram in Figure 10, both lists are then sent to the auditors as inputs to the verifier.[12] We recommend deleting L_{votes} both from the return values of ProcessPlaintexts and from the inputs to the verifier, because the factorizations need to be computed by the verifier in Step 6 of [VerSpec, Algorithm 4.3] (VerifyProcessPlaintexts) as part of the verification. Sending this list as an additional input to the verifier creates unnecessary redundancy and an additional verification step in Step 11 of [VerSpec, Algorithm 4.3]. We also recommend sending $L_{\text{decodedVotes}}$ in its pure form to the auditors, not encoded as an XML file called "`evoting-decrypt.xml`", because the XML format should not appear in the description of the cryptographic protocol. Note that an implementation of Algorithm 3.1 (EncodeVotingOptions) is still missing (see remark in Subsection 3.3.1).

- The generation and verification of Schnorr proofs has been added to Algorithm 4.1 (GenKeysCCR) and Algorithm 4.7 (GenVerCardSetKeys), respectively. Corresponding changes in some domain classes of the crypto-primitives-domain component have already been commented in Appendix A.2.3. Generally, these changes are necessary to avoid so-called *rougue key attacks*, and therefore we support this extension. The Java code of these algorithms has been adjusted accordingly. The generation of a Schnorr proof has also been added to Algorithm 4.10 (SetupTallyEB), but the further processing of the additional return value $\boldsymbol{\pi}_{\text{EB}}$ remains unclear. According to Figure 7, at least, $\boldsymbol{\pi}_{\text{EB}}$ is not sent to any other party. However, in Table 15 we found an additional entry for $\boldsymbol{\pi}_{\text{EB}}$ in the message "SetupComponentPublicKeys" from the setup component to all other parties (including the auditors). In Table 3 of [VerSpec, Version 1.1.0], this message is also listed, but its content has not been updated. Nevertheless, Verification 1.04 (VerifyKeyGenerationSchnorrProofs) includes corresponding proof verification steps. Therefore, it seems that the problem encountered here is an inconsistency introduced while introducing a necessary security extension. This problem needs to be addressed by fully clarifying the implemented solution. This includes explaining the purpose of sending $\boldsymbol{\pi}_{\text{EB}}$ to the online and tally control components and giving a justification of letting the untrusted setup component performing the proof verifications.

[11] We found a shell script `genEncryptionParametes.sh` in the folder `tools`, which indicates that the encryption parameters are generated by a separate process, possibly invoked by a human administrator. The problem with this shell script is that it refers to an outdated file config-cryptographic-parameters-tool-0.15.3.1-SNAPSHOT.jar. Besides, the file name contains a typo ("`Parametes`").

[12] In the protocol messages listed in [SysSpec, Table 17] and [VerSpec, Table 5], the list L_{votes} is linked to the context data ("`decoded votes`", `ee`, `bb`), but the list contains *encoded* votes.

- Issue #5 on the GitLab's project page describes a vulnerability of the implemented USB import/export functionality. The problem is that the insecure implementation does not validate the files that are copied, which may allow the files in the SDM directory to be overwritten by an attacker. An entry in the CHANGELOG.md file states that an allow list has been implemented to restrict the copying of files from a USB stick. Since this is not a cryptographic topic, we can only confirm that corresponding changes were made to the responsible Java class `OperationsController`, but we cannot confirm whether these changes are sufficient to fully mitigate this problem.
- Issue #35 on the GitLab's project page describes an improper sanitization of HTTP query parameters. Corresponding changes were already made in Version 0.15.2.3 on July 26. Since this is again a non-cryptographic matter, we can confirm that changes were made to the responsible Java class `HttpRequestSanitizer`, but we cannot confirm whether these changes are sufficient to fully mitigate this problem.
- Another documented change is the introduction of an additional election output in form of an XML document called "eCH-0110_xx.xml". The tally control component creates this file based on the decoded votes and sends it to the auditors. Some additional Java classes have been introduced for this purpose, most notably a service class called `TallyComponentEch0110Service`. We are not convinced that this extension is a proper answer to our objection discussed in Subsection 3.1.3, because the information contained in this file is highly redundant, and therefore its purpose remains questionable. Furthermore, we believe that the XML format should not appear in a cryptographic specification.
- Along with the introduction of parallel stream processing the crypto-primitives component (see discussion in Appendix A.2.1), the same performance optimization has been introduced at various places of the e-voting component. A total of 12 Java source files are affected in the modules secure-data-manager/backend and control-components. This modification is not documented in the project's CHANGELOG.md file.
- There is another undocumented change in Algorithm 5.1 (GetKey), which affects multiple lines of pseudocode. Without giving any explanations, the exact purpose of this change remains unclear. To the best of our understanding, it addresses the fact that Argon2id requires a salt. In our remarks on Algorithm 4.8 (GenCredDat) in Subsection 3.3.2 and Algorithm 5.1 (GetKey) in Subsection 3.3.3, we have discussed the misalignment between specification and code in this matter, so the observed change in Algorithm 5.1 seems to be the necessary patch of the specification. The lack of clarity comes from the problem of splitting $\texttt{VCks}_{\texttt{id,combined}}$ correctly into its components. This is actually a classical serialization/deserialization problem, which could be solved more easily with the right abstractions.

The list of changes in the CHANGELOG.md file contains two additional entries related to the eCH-0110-4-0.xsd file, but we consider them less important from the perspective

of the cryptographic protocol. Some other observed changes are the result of cleaning up the code without changing it or are related to updating dependencies to third-party libraries.

A.3.2. Redefined and Usage of pTable

In the system specification, the biggest change introduced in [SysSpec, Version 1.1.0] is the redefinition and handling of the primes mapping table pTable and the derivation of the final election result. We have already discussed this topic at several places. In Subsection 3.3.3, we recommend that a list of selected voting options should be given as input to CreateVote, not a list of prime number encodings of the selected voting options, and in Subsection 3.1.3, we object that the authenticity of the pTable object is not guaranteed and that the protocol stops with the factorization of the votes instead of interpreting them as an election result. Finally, in Subsection 1.4, we recommend removing pTable completely from the protocol, because it contains no relevant information that otherwise could not be determined deterministically.

Under certain circumstances, the protocol as specified in the August release allowed an attacker to completely invert the final election result. In the updated specification document of the October release, pTable is still present, but as stated in the project's CHANGELOG.md file, pTable is now included in the auxiliary input $\mathbf{i}_{\mathsf{aux}}$ of the zero-knowledge proofs generated by the client during vote casting (see Lines 10 and 11 of Algorithm 5.2, CreateVote). Therefore, all parties verifying these proofs using either Algorithm 5.3 (VerifyBallotCCR) or Algorithm 6.4 (VerifyVotingClientProofs) from [SysSpec], or Algorithm 4.1 (VerifyOnlineControlComponentsBallotBox) from [VerSpec], must share the same pTable to verify the proof successfully.

In our opinion, this is generally an elegant approach to guarantee a consistent view across all involved parties, because it delegates the consistency check to the trusted control components and the auditors. The voting client just binds its view to the cast vote and the trusted parties verify whether their own view corresponds to it or not. Note that the same approach has been used already in the August release to coordinate the views between Algorithm 4.8 (GenCredDat, called by the setup component) and Algorithm 5.1 (GetKey, called by the voting client), but there the motivation is less clear, because the setup component is untrusted (for simplicity reasons, we recommend removing it).

Unfortunately, the implemented solution only guarantees a consistent view of the pTable object, but it does not create a consistent view of the overall election context, which includes the correct interpretation of the voting options v_i contained in pTable. Note that the definition of pTable has slightly changed in the October release. Previously, voting options were arbitrary strings $v_i \in \mathbb{A}^*_{UCS}$, but now they are limited to non-empty strings $v_i \in \mathcal{T}_1^{50}$ of maximally 50 characters from the alphabet $\mathcal{T}_1 = \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{0}, \ldots, \mathtt{9}, \mathtt{-}, \mathtt{\_}\}$, see [SysSpec, Secton 3.4.2]. Unfortunately, no further details are provided about the

format, properties, and interpretation of these strings. In this important matter, the current protocol specification is completely underspecified.

In our attempt to correctly understand the meaning of the voting option strings $v_i \in \mathcal{T}_1^{50}$, we observed in [SysSpec, Table 15] and [VerSpec, Table 3] a new entry for a message called "SetupComponentConfig" (with content configurationXML), which is sent by the setup component to the tally control component and the auditors. Unfortunately, this message is not shown in any of the protocol diagrams. To the best of our understanding, it should be sent along with pTable. Note that in the protocol diagram in [SysSpec, Figure 7], instead of sending pTable to the tally control component, who requires pTable in Algorithm 6.7 (ProcessPlaintexts), it is sent to the CCMs, who do no require it. Regarding the content of the file configurationXML, the specification documents do not give further information, except for the remark in [VerSpec, Section 3.2] stating that the file is signed and for the additional input in [VerSpec, Algorithm 4.4] (VerifyTallyFiles). By inspecting the provided test datasets in the verifier GitLab repository (files electionEventContextPayload.json and configuration-anonymized.xml), we conclude that the voting options v_i are unique identifiers and that the file configurationXML defines its interpretation for the current election.

Assuming that our interpretation of the values v_i as unique identifiers for the file configurationXML elements is correct, the problem that arises from this solution is the fact that the setup component, who signs configurationXML and sends it to the offline parties (tally control components and auditors), is not trustworthy with regard to universal verification. Therefore, a dishonest setup component under adversarial control may possibly send a modified configurationXML file to the online parties (CCRs, voting clients), for example one in which the identifiers for the voting options `YES` and `NO` in two simultaneous referendums are switched. Currently, such an attack cannot be detected by the auditors conducting the universal verification, even if additional steps for decoding the voting options have been added to [SysSpec, Algorithm 6.7] (ProcessPlaintexts) and [VerSpec, Algorithm 4.3] (VerifyProcessPlaintexts).

The simplest workaround to mitigate this problem in the implemented approach discussed above would be to include both pTable and configurationXML as auxiliary inputs $\mathbf{i}_{\mathsf{aux}}$ for the zero-knowledge proofs, even if we generally do not recommend using XML-encoded data as inputs to a hash function (XML encodings are not unique). This would guarantee a consistent view of both the encoding (pTable) and its interpretation (configurationXML) across all involved parties.

This said, we would like to emphasize that the problem discussed here seems to be even more profound. In our 2021 report, we have already explained that several important parameters were missing in the abstract election event model, and we have given some recommendations for improving it (see [HKLD22b, Section 2.4.7]). Unfortunately, we did not observe much progress in this matter, because a well-specified election context is still missing in the current specification document (see remarks in Subsection 3.1.2). Note that the problem described here is very closely related to the point raised in our last year's report, namely that a comprehensive set of election parameters, which includes all

relevant information about the current election event, needs to be defined as a common starting point for all parties involved in the protocol. In CHVote, for example, the variable $EP = (U, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{c}, \mathbf{k}, \mathbf{E},)$ fully specifies all relevant parameters of an election event, including unique textual descriptions of the voting options [HKLD22a, Section 6.3.2]. To impose a consistent view during a protocol execution, every message is unambiguously linked to either the full set of parameters EP or a voter-specific subset of parameters VP_v [HKLD22a, Section 7.4]. Without defining the election context in such a clear way, the struggle of creating a consistent view of the current election will remain. To the best of our understanding, the parameters included in EP can be derived deterministically from the given eCH document specifying the election event.

To complete our remarks on this topic, we recommend implementing the following minor improvements:

- The comment "*Matching order between encoded and actual voting options*" in Algorithm 4.3 is not necessary.
- The comment "*Links the zero-knowledge proofs to the voting client's view of the* pTable*, which maps the actual to the encoded voting options.*" should be removed from Algorithm 5.2. Such clarifying explanations should be given in the regular text.
- We found different formal definitions of pTable, first as a pair $(\tilde{\mathbf{v}}, \tilde{\mathbf{p}})$ of vectors, and second as a vector $((v_0, \tilde{p}_0), \ldots, (v_{n-1}, \tilde{p}_{n-1}))$ of pairs. Given its domain $(\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \backslash g)^n$, the latter definition is the correct one. Using both notation interchangeably is abusive.
- It is confusing to have variables v_i, but vectors $\tilde{\mathbf{v}} = (v_0, \ldots, v_{n-1})$.
- Sometimes, pTable is part of the context (for example in VerifyBallotCCR), sometimes $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{p}}$ are part of the context (for example in CreateVote), and sometimes $\tilde{\mathbf{v}}$ and $\tilde{\mathbf{p}}$ are inputs (for example in GenVerDat).

Besides these remarks and recommendations, the implementation is aligned with the specification.

A.3.3. JavaScript Client

The voter-portal component has been completely rewritten in Angular, eliminating one of our main criticisms on the client-side (see Subsections 1.4 and 3.3.3 and [HKLD22c, Section 2.5]). Angular Version 13 was selected, which guarantees long-time support until May, 2023. The migration from AngularJS to Angular is not explicitly mentioned in the CHANGELOG.md file, but it can be regarded as summarized under "*Updated dependencies and third-party libraries*". In Table 3, this migration is one of the main reasons for the large number of changed source files.

As a result of this update, we evaluated the integration of the cryptographic protocol algorithms. In our first examination, we excluded this aspect and focused solely on the algorithms themselves, as we knew that the voter-portal will have to be rewritten in the course of the pending migration. The top-level protocol algorithms are implemented in the Maven module voting-client-js, which is also the interface between the cryptographically relevant and non-relevant parts of the code. Note that for some algorithms, voting-client-js depends on both the new crypto-primitives-ts library and the old cryptolib-js library from the former Scytl system. The latter is mainly used for voter authentication.

The cryptographically relevant code of the voting client, that is the voting-client-js with all its dependencies, is entirely executed in a dedicated *Web Worker* environment of the voter's web browser, i.e., alongside with the main Angular application. This general architecture is a good strategy for preventing unintentional interferences between different parts of the code bases. If a web client is implemented using a framework such as Angular, and therefore depends on numerous third-party libraries, a strict separation of the cryptographically relevant code helps to reduce the risk of generic attacks from contaminated or flawed third-party dependencies imported from npm. We have repeatedly expressed our concerns about the simplicity and impact of such attacks, for example in Subsection 2.2.7 and in [HKLD22c, Section 2.7]. In one of the simplest attacks, the adversary takes full control over the cryptographic randomness source with a single line of infiltrated JavaScript code. Without isolating the execution of the cryptographically relevant code from the rest of the system, this line of code could be injected though any of the imported third-party libraries.

To obtain a complete picture of the cryptographic core that runs in the separate Web Worker environment, we built the system and analyzed the resulting ov-api.min.js file. The content of this file is exactly what is executed by the Web Worker. From building this file, we learned that it consists of almost 100'000 lines of JavaScript code. Some of the code lines are minified versions of complete libraries on a single line of code. To realize that the quantity of code included in this file is enormous was quite a surprise, because we had expected a slim cryptographic library in which every code line can be explained and justified. In the resulting file obtained from the build process, quite the opposite is true. Unfortunately, this turns the benefits of the selected approach with a dedicated Web Worker to the opposite.

The file ov-api.min.js consists of several external cryptographic libraries with similar functionalities. As a result, it contains multiple implementations of the same cryptographic standard algorithms such as AES or SHA256, and it includes three different full-featured big integer implementations jsbn[13], vts[14], and bn.js[15]. It also contains old and unsup-

[13] https://github.com/andyperlitch/jsbn
[14] https://github.com/verificatum/verificatum-vts-ba
[15] https://github.com/indutny/bn.js

ported libraries, for example crypto-js[16] (last updated in August 2013) or json-sans-eval[17] (last updated in 2009), libraries from completely different application areas, for example bitcoinjs-lib[18], and different versions of the same library, for example bn.js@4.12.0 and bn.js@5.2.1. Figure 9 gives an overview of the general dependency structure of the whole voting-client-js component.

We have repeatedly recommended to reduce the amount of third-party libraries, for example in [HKLD22c, Section 2.5], so we were quite surprised to see that the situation has not improved much in the meantime, not even in one of the most security-critical components of the system. This gives a poor overall impression and still makes the voting client highly vulnerable to generic attacks through third-party dependencies.

```
voting-client-js (8/59)
├─ core-js
├─ crypto-browserify (11/38) ⇒ added by the build system
├─ cryptolib-js (2/2)
│  ├─ node-forge
│  └─ sjcl
├─ crypto-primitives-ts (3/3)
│  ├─ hash-wasm
│  ├─ jssha
│  └─ vts
├─ jsrsasign (4/23)
│  ├─ jsbn
│  ├─ cryptojs
│  ├─ bitcoinjs-lib (8/19)
│  └─ json-sans-eval
├─ lodash
├─ q
└─ xhr2
```

Figure 9: Dependencies of the voting-client-js component. The first number given in parentheses indicates the number of direct dependencies, and the second the total number of dependencies (without development dependencies). We computed these numbers using the CLI command "`npx npm-remote-ls --development=false <library>`".

An interesting technical aspect, which we observed while inspecting the updated crypto-primtives-ts library, is the fact that the `window` object is accessed at the most critical place (while generating random bytes, see Figure 10). However, the `window` object does not exist inside a Web Worker environment. Only thanks to the build system, the

[16] `https://code.google.com/archive/p/crypto-js`
[17] `https://code.google.com/archive/p/json-sans-eval/`
[18] `https://github.com/bitcoinjs/bitcoinjs-lib`

```
/**
 * Provides cryptographically strong random values.
 *
 * This pseudo-random number generation relies on the availability of basic cryptography features available in the current context.
 * It looks for the available crypto module and random method in this order:
 *  - WebAPI Crypto.getRandomValues()
 *  - NodeJS Crypto.randomBytes()
 *  - No available module, it throws an Error
 *
 * @param size The number of random values to generate. Must be non null and strictly positive.
 * @return A TypedArray filled with random values
 */
public static genRandomBytes(size: number): ImmutableUint8Array {
        checkNotNull(size, "The size can not be null");
        checkArgument(size > 0, "The size must a be a positive number greater than 0.");

        // User agent WebAPI Crypto
        if (typeof window !== "undefined" && window.crypto && window.crypto.getRandomValues) {
                return ImmutableUint8Array.from(window.crypto.getRandomValues(new Uint8Array(size)));
        }

        // NodeJS Crypto module fallback
        if (crypto && crypto.randomBytes) {
                return ImmutableUint8Array.from(crypto.randomBytes(size));
        }

        throw new Error(`Neither "Web API Crypto" nor "NodeJS Crypto" is available`);
}
```

Figure 10: Accessing the `window` object in secure_random_generator.ts.

NodeJS fallback works also in the browser context, because it resolves the dependency and includes the module crypto-browserify[19] (and thus adds a large number of unnecessary additional dependencies into the built library). This gives the impression, that not even the developers at Swiss Post are fully aware of what is really going on inside the cryptographic core of the voting client.

A.4. Verifier

Many changes in the updated verifier component reflect the changes of the October release discussed for the e-voting component. Our analysis in Appendix A.3 already contains several references to both the verifier specification and the source code. The following list summarized corresponding topics of conducted modifications (they are all listed in the project's CHANGELOG.md file):

- A new list $L_{\text{decodedVotes}}$ of decoded votes generated by the tally component is given as additional input to the verifier. It is sent to the verifier in a message file called tallyComponentVotesPayload.json. This list is unwrapped in Verification 2.11 (VerifyTallyControlComponent) and its consistency is checked in Line 11 of Algorithm 4.3 (VerifyProcessPlaintexts). The implementation has been adjusted accordingly.
- Introducing Schnorr proofs to Algorithms 4.1 and 4.7 creates an additional component to the message "SetupComponentPublicKeys" sent from the setup compo-

[19] `https://www.npmjs.com/package/crypto-browserify`

nent to the auditors and requires additional verification steps in Verification 1.04 (VerifyKeyGenerationSchnorrProofs). The implementation has been adjusted accordingly.

- The inclusion of pTable as an additional auxiliary input to some zero-knowledge proofs requires the adjustment of corresponding verification algorithms. Furthermore, pTable needs to be sent to the auditors to allow them performing the verifications in Algorithm 4.1 (VerifyOnlineControlComponentsBallotBox). The message "SetupComponentPublicKeys" has been extended for that purpose.
- Due to the general migration from 112 to 128 bits security of security, an additional Require statements has been added to Verification 1.01 (VerifyEncryptionParameters). This seems to be correct, but it disables elections executed in *esting-only* or *legacy* mode (for example for testing purposes) from being verified correctly. We are not sure if this is intended.

The file CHANGELOG.md also mentions the inclusion of some additional consistency checks for the general verification procedure VerifyTally and an improved input validation for Algorithm 3.1 (VerifyEncryptedPCCExponentiationProofsVerificationCardSet), but without further specifying which ones. We observed, that a new Algorithm 4.4 (VerifyTallyFiles) has been introduced for checking the consistency of the output XML files "`evoting-decrypt.xml`" and "eCH-0110_xx.xml". It is called by Verification 2.11 (VerifyTallyControlComponent). We assume that this is what is meant with the "*inclusion of some additional consistency checks*" in the CHANGELOG.md file. A few other changes are listed, for example the migration of JKS keystores to standard PKCS12 keystores, the updating of the dependencies to third-party libraries, or the fixing of the incorrect ordering when reading files from the file system, but we consider them less important for our analysis.

To describe the outcome of our analysis of the verifier component more systematically, we add two more subsections with further details on some findings relative to the updated specification document and source code. Independently of our remarks on the technical details, our general impression that many aspects of the verifier are still not sufficiently well specified remains, and also that the specification still seems to lag behind the implementation. A thorough and systematic alignment analysis is therefore still not possible, because simply too much room for interpretation and ambiguities exist in the current specification.

A.4.1. Verifier Specification

To illustrate the above-mentioned lack of preciseness and disambiguation in the verifier specification, we provide the following list to summarize all the statements included in the pseudocode algorithms that are given in textual form:

- Verification 1.21: VerifyEncryptedPCCExponentiationProofs

  - Line 3: Context and Input for verification card set $\texttt{vcs}_i$ and control component j
- Verification 1.22: VerifyEncryptedCKExponentiationProofs
  - Line 3: Context and Input for verification card set $\texttt{vcs}_i$ and control component j
- Verification 2.01: VerifyOnlineControlComponents
  - Line 2: Extract the key-value map of the verification card public keys **KMap** from the Setup Component Tally Data
  - Line 3: Prepare the *Context* including the election event context
  - Line 4: Prepare the *Input* containing **KMap**, the first control component's ballot box, and the online control component shuffles
  - Line 5: Context and Input for ballot box $\texttt{bb}_i$
- Algorithm 4.1: VerifyOnlineControlComponentsBallotBox

  Line 1: Extract the key-value map of verification card IDs to encrypted, confirmed votes from the first Control Component Ballot Box $\mathbf{vcMap}_1$
- Verification 2.11: VerifyTallyControlComponent
  - Line 2: Prepare the *Context* containing the parameters from the election event context including the primes mapping table pTable
  - Line 3: Extract the partially decrypted votes $\mathbf{c}_{\mathsf{dec},4}$ from the last control component's shuffle
  - Line 4: Extract $(\mathbf{c}_{\mathsf{mix},5}, \pi_{\mathsf{mix},5}, \mathbf{m}, \pi_{\mathsf{dec},5})$ from the Tally control component shuffle
  - Line 5: Extract the list of selected encoded voting options L_{votes} and selected actual voting options $L_{\mathsf{decodedVotes}}$ from the Tally control component votes
  - Line 7: Context and Input for ballot box $\texttt{bb}_i$
  - Line 9: Context and input as specified
- Algorithm 4.4: VerifyTallyFiles
  - Line 1: Aggregate the decoded voting options from all ballot boxes
  - Line 2: 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.

Given these excerpts from the pseudocode, it is clear that this is not sufficient for reviewers of the given verifier implementation or for developers of an independent verifier. Possibly the most extreme case is the new Algorithm 4.4 (VerifyTallyFiles), which in its current form leaves all the necessary details open. By introducing further underspecified algorithms, the problem discussed in Subsection 3.4 has been worsened.

The related lack of accuracy concerning the completeness, authenticity, consistency, and integrity checks of the verifier's input data has been accentuated by introducing three additional inputs in XML format: "configuration XML", "evoting decrypt XML", and "eCH 0110 XML". The specification contains references to corresponding XML schema documents (.xsd files), which can be used to validate the given files, but otherwise the content

and purpose of these documents remains largely unspecified.[20] Furthermore, as explained in [VerSpec, Sections 3.2 and 4.2], all three XML files are signed by either the setup component or the tally control component, but for the signature generation and verification, only a high-level description is given at the end of [VerSpec, Sections 3.2]. Generally, we do not recommend using the XML format in a cryptographic specification document.

To conclude this section, we provide a list of unrelated observations made during our analysis of the updated verifier specification:

- pTable has been moved from the "SetupComponentTallyData" to the "SetupComponentPublicKeys" message, but the reason for this remains unclear (both messages are sent at the same time).
- The leading text for Verification 1.04 contains a word repetition: "*that that*".
- In [VerSpec, Section 4.5], the list of evidence checks includes an additional entry "*Verify the Tally control component's generation of the tally files*", but the exact checks of this verification step remain unclear.
- In Algorithm 4.1 (VerifyOnlineControlComponentsBallotBox), the pTable is included in the election event context. However, it seems as if nothing is really done with it.

A.4.2. Verifier Implementation

As already discussed in the previous subsection, performing a systematic code analysis is much more difficult in the case of an inexact specification document that leaves too much room for interpretation. Compared to the of other component of the system, our code analysis of the verifier component has therefore been less accurate and not as comprehensive. The following list summarizes some general observations and provides feedback on certain specific points:

- With regard to the general topics discussed in the beginning of this section, we can confirm that corresponding code areas have been updated properly.
- In the tools section of the verifier-backend module, two new mapper classes called `ContestResultsMapper` and `DeliveryMapper` have been added in the new version. They both provide an impressive amount of mappings for verification. Unfortunately, the purpose of these classes is not very well documented, so they are only comprehensible when reading their code and the document structures they rely on. Here, we would hope the get some more clarifications from the specification.

[20] The XML schema "eCH 0110-4-0.xsd" from 2018 is referenced, but a new version "CH-0110-4-1.xsd" (see `https://www.ech.ch/de/ech/ech-0110/4.1`) is available since 2020. The implication of using the older version lies beyond our expertise and is noted as an informative fact.

- The updated implementation contains six different `HashableEchxxxxFactory.java` interfaces. Since a description of the eCH standards is explicitly excluded from the verification specification, we cannot independently audit their correctness and integrity without obtaining more information about this aspect. Again, we would hope to find such information somewhere in the specification.
- There is a new consistency check that verifies the correct naming of files. The class responsible for this check is called `VerifySetupFileNamesConsistency`. However, its purpose remains unclear, because it seems to be a very implementation-specific internal check. Unfortunately, no information is provided in the underlying specification.
- The internationalization of the resources has started. We found corresponding files for German, English, French, and Italian:
  - `resources_de.properties`,
  - `resources_en.properties`,
  - `resources_fr.properties`,
  - `resources_it.properties`.

  However, many dialogues are still only available in English.

B. Addendum-2: November and December Releases

During our analysis of the October release, major updates of both the specification documents and the code were announced on November 4, 2022. On gitlab.com, the updates were already available one week earlier, on October 27 (code) and October 31 (documents), respectively. In the sequel, we will refer to this version as the *November release*. Many of the changes made in the November release address the issues listed in Sections 2 and 3 of this report. As already mentioned, a draft of this report was given to the Swiss Post in the second half of September, which means that the period for making these changes was approximately five weeks.

Together with the November release, we received an updated project roadmap with an announcement of the subsequent release for early December. On December 9, 2022, the *December release* was published on the project's web site on gitlab.com. It includes new versions of all specification documents and code repositories. Since we received these updates at an early stage of evaluating the November release, we have agreed with the Federal Chancellery to analyze the November and December releases together in a single evaluation round. The results of our evaluation are described in the following subsections.

B.1. Overview of Changes

The following list gives an overview of all documents updated in the December release on December 9, 2022. These are the documents that we considered for the final version of this report:

- [CryptPrim] *Cryptographic Primitives of the Swiss Post Voting System – Pseudocode Specification*, Version 1.2.0, Swiss Post Ltd., December 9, 2022
- [SysSpec] *Swiss Post Voting System – System Specification*, Version 1.2.0, Swiss Post Ltd., December 9, 2022
- [VerSpec] *Swiss Post Voting System – Verifier Specification*, Version 1.3.0, Swiss Post Ltd., December 9, 2022
- [ProtSpec] *Protocol of the Swiss Post Voting System – Computational Proof of Complete Verifiability and Privacy*, Version 1.1.0, Swiss Post Ltd., December 9, 2022
- [ArchDoc] *E-Voting Architecture Document*, Version 1.2.0, Swiss Post Ltd., December 9, 2022

As in earlier updates, it was inherently difficult to locate the changes made to these documents, since they are given as PDF-files. However, similar to the October release, they all contain a revision chart with links to the *change logs* on gitlab.com. Additionally, we received special versions of [CryptPrim], [SysSpec], [ProtSpec], and [ProtSpec], which

highlight the changes made to these documents since the October version (to the best of our knowledge, these documents are not publicly available). While these documents were very useful for tracking the changes in the November release, we received them for the December release only on December 23. At that time, we had already manually tracked the changes in most parts of the documents.

Regarding the software, the first major release 1.0.0.0 from October was updated to 1.1.0.0 in November and to 1.2.0.0 in December (with a few minor internal releases 1.1.0.x in between). A complete overview of the latest versions released on gitlab.com is shown in Figure 11. Note that the version numbering of the verifier is still slightly different (the October release 1.1.0.0 has been updated to 1.2.0.0 in November and to 1.3.0.0 in December). On December 23, we were notified that a "*patch 1.2.1.0 of the e-voting system*" and a "*patch 1.3.1.0 of the verifier*" had been released on December 22, but that "*the patch [...] does not include any changes that affect the implementation of the cryptographic protocol*". Given the time constraints at such a late stage of our assessment, we were unable to look at the latest update and to verify the above claim (we only observed that 64 files were changed in the e-voting system and 11 files in the verifier). Our analysis is therefore restricted the versions released on December 8.

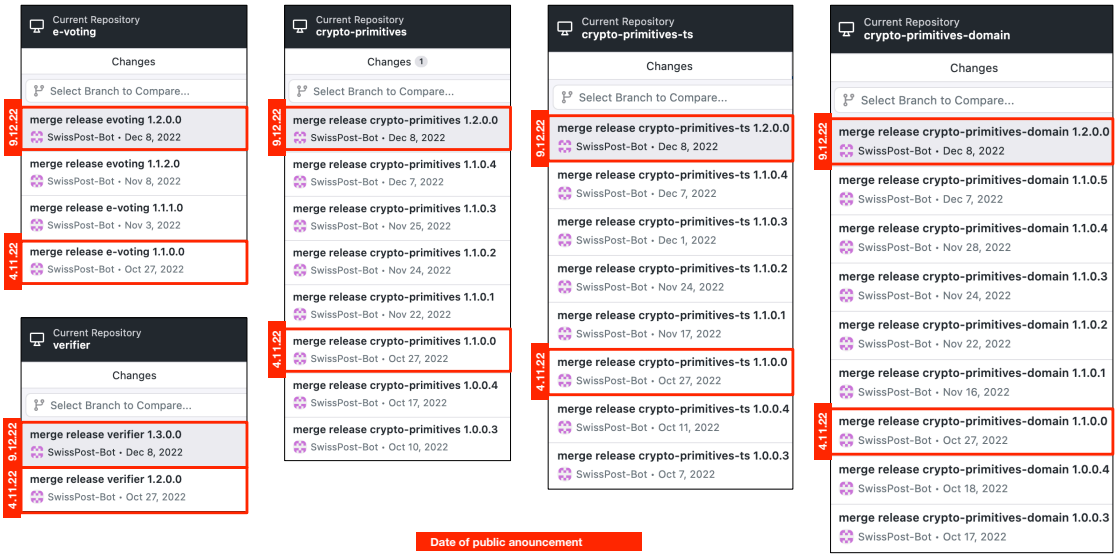


Figure 11: Commit histories of the GitLab projects with dates of public announcements of the November and December releases. It does not include the latest "patches" from December 22.

The amount of code changes since the October release is reflected by the number of source code files appearing in at least one of the GitLab commits from Figure 11. An overview of these numbers is given in Table 4 (changes in test files are ignored). We evaluated these numbers again to obtain a rough overview of the code changes and as a point of orientation for getting our analysis started at the right places. Generally, the large number of modified files seems to indicate that the state of the project is still

relatively unstable.

Maven Module	.java	.js	.ts	.json	.html	.xsd	.xml	.sql	.properties
crypto-primitives	61	–	–	–	–	–	–	–	–
crypto-primitives-domain	26	–	–	–	–	–	–	–	–
crypto-primitives-ts	–	–	13	2	–	–	–	–	–
command-messaging	2	–	–	–	–	–	–	–	–
control-components	51	–	–	–	–	–	–	1	1
cryptolib	12	–	–	–	–	–	–	–	–
cryptolib-js	–	–	–	2	–	–	–	–	–
domain	228	–	–	–	–	1	1	–	–
secure-data-manager/backend	139	–	–	1	–	1	1	–	4
secure-data-manager/frontend	–	9	–	10	6	–	1	–	–
tools	4	–	–	–	–	–	–	–	–
voter-portal	–	1	53	9	34	–	–	–	–
voting-client-js	–	14	–	2	–	–	–	–	–
voting-server	65	–	–	–	–	–	–	1	1
verifier-assembly	–	–	–	–	–	–	2	–	–
verifier-backend	97	–	–	1	–	–	2	–	1
verifier-frontend	–	1	2	4	1	–	–	–	–
verifier-protocol	23	–	–	–	–	–	–	–	–

Table 4: Number of files changed since the October release (without test files).

Compared to Table 3 in Appendix A for the October release, the picture in Table 4 is very similar. It shows that the largest number of changes have been made again to the modules secure-data-manager (backend), voter-portal, and domain of the e-voting component (as in the October release, the changes in the module domain are mostly auto-generated by JAXB and marked as such). Note that an increased amount of changes has been made to the crypto-primitives component. This is mainly a consequence of the improvements made in response to our findings listed in Subsection 3.2.

In the remaining of this addendum section, we summarize the findings of our last evaluation round. As in Section 3 and Appendix A, we structure our analysis according to the changes made in each of the three main system components. In Appendix B.5, we recapitulate the major conclusions of our assessment from a big picture's perspective.

B.2. Cryptographic Primitives

Since the beginning of our assessment, the modules crypto-primitives, crypto-primitives-ts, and crypto-primitives-domain have been the most stable components of the whole system.

The correspondence between documentation and code was already relatively high when we first looked at it in June, 2022. However, in our thorough analysis of all algorithms and their implementations in Java and TypeScript, we encountered numerous issues to improve and provided a long list of recommendations. In the updates that we received with the November and December releases, we observed that many of the issues and recommendations from Subsection 3.2 have been addressed, especially those related to deviations between the pseudocode and Java-code algorithms. To provide an overview of the improvements, we systematically inserted comments of the form [updated in November release] or [updated in December release] whenever possible. Given these improvements, the alignment between documentation and code has been further increased for the crypto-primitives component.

This said, we also observed that many of our findings and recommendations have not been considered. From the general problems discussed in Subsection 3.2.1, for example, most of the listed items have not been addressed at all. The same holds for most of the simplifications proposed in Subsection 1.4 (one exception is the removal of the algorithm IsProbablePrime and its sub-algorithms from [CryptPrim] in the November release). This is very unfortunate, because it means that the evident large potential for improving the code quality has not been fully exploited. We assume that within the given tight time constraints of the project roadmap, simplifications and code quality issues were not given the highest priority. However, we hope that our recommendations will be considered at a later stage of the product's life cycle.

We also observed that the minor algorithmic issues listed in Subsection 3.2.2 to 3.2.8 have not been addressed in a very systematic manner. For example, at several places in the pseudocode algorithms, we detected abusive notations like $h\|\mathbf{v}$ instead of $\langle h\rangle\|\mathbf{v}$ in Line 3 of Algorithm 4.9 or $v \leftarrow v \cup g_i$ instead of $v \leftarrow v \cup \{g_i\}$ in Line 9 of Algorithm 8.6. We made corresponding remarks in the tables of Subsections 3.2.3 and 3.2.7, respectively, but while the problem in Algorithm 8.6 has been addressed in the December release, it still persists in Algorithm 4.9. We agree that these are mostly tiny problems, but we would have expected a more thorough and comprehensive consideration of our findings.

To conclude the point raised in the previous paragraph, we must note that on November 22 and on December 22, 2022, we received from Swiss Post two response documents addressed directly to us, in which they justify their decision not to address some of our findings. These documents contain justifications like "*we analyzed your suggestion but prefer to keep the current approach*", "*we consider this a stylistic preference*", or "*our position is that we consider [...] a cryptographic bad practice*". Clearly, subjective statements like this are not very persuasive in all cases. In a few cases, we would have liked to express our objection, but given the tight time constraints of our assessment, we decided to just leave our discussion in Subsection 3.2 untouched, by which we express our opinion that the problems still persist in the current version.

B.3. E-Voting

Like in the October release, the largest number of changes has been made to the E-voting component. Changes have been made to both the documentation and the code. As one can see in Table 4, almost all Maven sub-modules are affected by the code changes. Generally, we did not expect to encounter several new features and discussions of new aspects at this late stage of the development process, where one would expect that most efforts are put into stabilizing the implementation, improving its quality, and finalizing the code base. The same happened already in the October release, where a completely new voting client was added to the code base. Again, this does not give the impression of a stable and robust system that will soon reach the status of a market-ready product. We understand that the whole system is very complex and its development requires enormous efforts, but we don't think that numerous last-minute changes under time pressure are the right answer for achieving the security and code quality goals.

Some of the security-relevant enhancements that have been made to the specification document [SysSpec] and the code are the following (from either the November or December releases):

- A new Section 1.5 on "*Context, State, and Input Variables*" has been added to the introductory section, and a new message *ElectionEventContext* has been added to the overview of the messages and signatures in Table 15 of Section 7.
- A new Section 3.6 about "*Write-Ins*" with six new Algorithms 3.7 to 3.12 has been added to the preliminaries section. Corresponding Java methods and JavaScript functions have been added to the modules secure-data-manager (backend) and voting-client-js, respectively.
- Additional information and explanations about the *authentication protocol* have been added to Section 5.1 (before Section 5.1.1).
- Additional information about handling votes submitted over conventional channels has been added to Section 5.2.4. This important aspect has not been discussed in earlier releases.
- A new Section 6.2.6 on "*Requesting a Proof of Non-Participation*" with a new Algorithm 6.8 has been added to the description of the tally phase (but not to the code base).
- A new Section 6.2.7 on "*Handling Inconsistent Views of Confirmed Votes*" with two new Algorithms 6.9 and 6.10 has been added to the description of the tally phase (but not to the code base).
- A new protocol party "*Canton*" has been added (silently) to the overview of the messages and signatures in Table 15 of Section 7. Similarly, a new trustworthy protocol party "*Dispute Resolver*" has been introduced in the new Section 6.2.7.

- As announced in the CHANGELOG.md file of the December release, new sub-folders `security/windows` and `security/linux` with corresponding `java.security` properties files have been added to the main project repository.

Many of the enhancements in the above list are not very well specified and explained. As such, they do not contribute much to the clarification of corresponding topics. We will discuss some of the topics in the following subsections and describe our observations and conclusions from our second assessment round. We also looked at all the minor changes made to the algorithms and inserted in Subsection 3.3 comments of the form [updated in November release] or [updated in December release] whenever possible.

B.3.1. Context, State, and Input Variables

As a response to our remarks about the context variables in Subsection 3.1.2 and our comments on immutability in Subsections 1.4, 2.2.5, 3.2.9 and 3.3, the discussion in the new Section 1.5 provides some additional, but mostly vague explanations about the design concept of this important topic. Our main point in Subsection 3.1.2 is the missing formal definition of *the context* in any of the available specification documents. We illustrated the problem by giving a comprehensive list of context variables that appear in at least one algorithm. This list demonstrates the complexity of the topic and reveals numerous problematical and inconsistent cases in the current implementation. We recommended introducing a more consistent concept, that can be applied systematically to both the documentation and the code. Given the vagueness of the explanations given in the new Section 1.5, our recommendation has not been implemented and the situation has not been improved.

A particular aspect that we discussed in Subsection 3.1.2 is the existence of side-effects in some of the given algorithms. By saying in Section 1.5 that the context may also contain stateful lists and maps, our remark has been taken into account in the documentation, but our concerns about this design decision have not been considered (we explicitly recommended to "*exclude data structures such as lists or maps from any context to ensure that all algorithms are free from side-effects*"). Therefore, there are still many algorithms that produce a side-effect over the context variables, and some algorithms even directly access the database (read and write). As already discussed, side-effects always create an additional complexity layer that makes the examination and testing of such algorithms much more difficult. This is why they should be avoided whenever possible in complex systems as a measure to optimize their code quality and robustness.

Another weak point of the discussion in Section 1.5 is the distinction between variables that are *invariant* and variables that are *different for each invocation*. Generally, we consider it good design strategy to make such a distinction, but only if it is defined and implemented in a rigorous and consistent manner, both in the specification and the code. Unfortunately, this is still not the case in the current release. For example in Algorithm 6.7, $\tilde{\mathbf{p}}$ is defined as a context variable (as part of pTable), but at the same

time, $\tilde{\mathbf{p}}_w$ (a sub-vector of $\tilde{\mathbf{p}}$) is defined as an input variable. In this particular case, the inconsistent implementation of this idea is obvious (we mention it, because $\tilde{\mathbf{p}}_w$ has been introduced as an additional input variable in the November release), but as already noted in Subsection 3.1.2, there are many other similar examples, for example variables that are sometimes inputs and sometimes context.

B.3.2. Write-Ins

The support for write-ins has a long history in the Swiss Post e-voting system. In our 2021 report, we criticized the half-hearted inclusion of this topic in both the documentation and the code. In the August and October releases that we received for the current evaluation, write-ins were no longer included as a feature of the system, but they were not eliminated entirely. From a technical perspective, the most important precondition for including write-ins are *multi-recipient ElGamal* encryptions [CryptPrim, Section 7], which can be used to attach write-ins efficiently to regular votes. Since the topic of multi-recipient ElGamal encryptions could have been dropped without write-ins, we proposed in Subsection 1.4 their removal as a potential simplification. However, we knew from [ResScope1, Section 4.3] that dropping write-ins was never an option.

In the November release, write-ins have been reintroduced with a new Section 3.6 in [SysSpec], six additional Algorithms 3.7 to 3.12, and two enhanced Algorithms 5.2 and 6.7. We evaluated this new section, the new and enhanced algorithms, and their implementations. Here are some observations from our analysis:

- The purpose of the separator symbol # at index (rank) 0 of the alphabet is unclear. Why should a string with the name of a write-in candidate contain such a symbol? And what if a malicious voting client uses this symbol as prefix on purpose?
- In Line 1 of Algorithm 3.9, there are two possible outcomes for $x \leftarrow \sqrt{y} \bmod p$, one in $\mathbb{G}_q$ and not it $\mathbb{G}_q$ (or equivalently, one smaller than q, and one between q and $p-1$). The pseudocode algorithm must specify which one is taken. In the implementation of the algorithm in the class `QuadraticResidueToWriteInAlgorithm`, we observed that the sqaure root smaller than q is selected, which is the correct choice. However, documentation and code are not aligned. [updated in December release]
- In the Algorithms 3.9 to 3.12, the context is missing for no obvious reason. This looks like if checking the domains of the input variables has been dropped for sub-algorithms (which is something we proposed as a potential simplification in Subsection 3.2.1), but then this is inconsistent with the implementation of all other sub-algorithms.
- Empty strings for write-ins are excluded for no obvious reason. How is an empty string different from one that contains nothing but spaces? Explanations are missing to justify this choice.

- The notation for the input $x \in \mathbb{Z}_q \notin \{1\}$ in Algorithm 3.10 is abusive, it should $x \in \mathbb{Z}_q \backslash \{1\}$. Similarly, the expression $\hat{\mathbf{s}} \leftarrow (\hat{\mathbf{s}}, \hat{s}_k)$ in Line 6 in Algorithm 3.12 is slightly abusive. The same abusive notation is used in Lines 12–14 of Algorithm 6.7.
- In the example given in Subsection 3.6.4, only two out of three possible write-ins have been selected. For the third one, $w_{\mathtt{id},2} = 1$ is selected as a default encoding, but this corresponds to the encoding of a string "␣" consisting of a single space with rank 1 in the alphabet. It is unclear if this is intentional or not, and if not, if this can cause a problem.
- According to Figure 8, it is clear that the voting client performs CreateVote (Algorithms 5.2) with input values obtained from the voter, in particular the vectors $\hat{\mathbf{p}}_{\mathtt{id}}$ (encoded votes) and $\mathbf{w}_{\mathtt{id}}$ (encoded write-ins). Here, corresponding calls of the encoding algorithms are missing, particularly a call to WriteInToQuadraticResidue (Algorithm 3.7) for every element of $\mathbf{w}_{\mathtt{id}}$. The current document gives the impression that WriteInToQuadraticResidue is never called (respecively that the human voter executes WriteInToQuadraticResidue, which is absurd).
- In ProcessPlaintexts (Algorithm 6.7), a new input $\tilde{\mathbf{p}}_w$ (write-in voting options) has been introduced, but no definition or explanations are given. We conclude from Algorithm 3.11 that the purpose of $\tilde{\mathbf{p}}_w$ is to define the subset of the prime number encodings in $\mathbf{p}_w$ that correspond to write-ins. But then it is something that defines the current election and therefore should be part of the election context (for example as part of pTable). Given its importance for obtaining the correct final result, the specification must be clarified. Note that the requirement $\hat{\delta} \leqslant l$ is in conflict with the requirement of DecodeWriteIns (Algorithm 3.12), which expects the first and the third argument to be equally long, i.e., the requirement should be $\hat{\delta} = l$.
- According to ProcessPlaintexts (Algorithm 6.7), all write-ins are decrypted, including the empty ones. This means that the unused write-in fields in a vote for regular candidates offer the possibility for a malicious voting client to mark the ballot, and that such marks become visible after the decryption. We discussed this problem already in our 2021 report on Scope 1 [HKLD22b, Section 2.4.8].
- Another problem with the write-ins occur in cases where multiple write-in elections are held jointly in a single protocol run. In such a case, it looks like assigning the write-ins in the vector $\mathbf{w}_k$ to the right question may not always be unique in the current implementation where $\tilde{\mathbf{p}}_w$ is the only additional election parameter. Possibly, the problem can be solved by defining a unique ordering, but such an ordering is currently not specified.

In the light of these remarks, we do not think that the write-ins topic has been treated and implementation with the necessary care. Generally, we believe that such a complex and security-critical topic should not be introduced as a new feature shortly before reaching an important milestone in the project roadmap. On the other hand, we couldn't find any weaknesses in the write-ins implementation that could interfere with the base protocol in an election without write-ins. In such cases, where $\hat{\delta} = 1$ limits the size of

the public multi-recipient ElGamal key to 1 and the length of corresponding ElGamal ciphertext tuples to 2, everything degenerates into a protocol run without write-ins, i.e., with an empty input vector $\mathbf{w}_{\mathtt{id}} = ()$ in CreateVote, an empty input vector $\tilde{\mathbf{p}}_w = ()$ in ProcessPlaintexts, and empty output list $L_{\mathsf{writeIns}} = \langle\rangle$ in ProcessPlaintexts. Therefore, at least for elections without write-ins, it seems that extended system provides the same security.

B.3.3. Voter Authentication

In our 2021 report, we already noted that the transmission of the verification card keystore $\mathtt{VCks}_{\mathtt{id}}$ from the voting server to the voting client was not properly specified. In Figure 8 of the current version, this is unfortunately still the case. The following statement from [SysSpec, Section 5.1] expresses this problem explicitly:

> *"We omit the voter's authentication to the voting server to retrieve the Verification Card Keystore* $\mathtt{VCks}_{\mathtt{id}}$ *and we assume that the voting client authenticates to the voting server prior to the SendVote phase."*

This statement is still present in the December release, but some additional explanations about the authentication process are now given in Section 5.1:

> *"The authentication protocol requires the voter to enter the correct Start Voting Key* $\mathtt{SVK}_{\mathtt{id}}$ *and, optionally, an additional authentication factor such as the date of birth."*

Unfortunately, this explanation does not precisely define the authentication process. Even worse, it raises several questions, because it looks as if the start voting key $\mathtt{SVK}_{\mathtt{id}}$, entered by the voter, is submitted to the untrusted voting server. We took the lack of information as a starting point to investigate this aspect more profoundly in the implementations of the voting client and voting server. Our expectation was to find, for example, that a hash value $\mathsf{hash}(\mathtt{SVK}_{\mathtt{id}})$ of the start voting key is submitted to the voting server as a key for selecting the right $\mathtt{VCks}_{\mathtt{id}}$ from a map. However, we learned that the implementation does something much more complicated, which is even in contradiction with the recently added statement quoted above. Here is a direct comparison between the specification and our findings about the implementation:

Specification

- The start voting key $\mathtt{SVK}_{\mathtt{id}}$ is generated by the setup component and passed to the printer.
- The printer prints $\mathtt{SVK}_{\mathtt{id}}$ and the voter receives $\mathtt{SVK}_{\mathtt{id}}$ on paper.
- The voter enters $\mathtt{SVK}_{\mathtt{id}}$ to start the voting process.

- Upon receiving the keystore $\texttt{VCks}_{\texttt{id}}$ from the web server, the voting client opens $\texttt{VCks}_{\texttt{id}}$ using $\texttt{SVK}_{\texttt{id}}$.

Implementation

- The *authentication key* is generated by the setup component using one of the two classes (depending on the entry for `auth.generator.type` in the property file resources/application-standard.properties):
  - `SimpleAuthenticationKeyGenerator` (uses $\texttt{SVK}_{\texttt{id}}$ as authentication key),
  - `SingleSecretAuthenticationKeyGenerator` (generates a new authentication key by selecting 24 random characters from an alphabet of size 32, which corresponds to a total of 120 random bits).
- The printer prints the authentication key and the voter receives it on paper.
- The voter enters the *initialization code* (synonym for *authentication key*).
- The voting client derives an *authentication ID* and a symmetric encryption key from the initialization code (see JavaScript code in authenticate.js):

$$\begin{aligned} \mathit{authenticationId} &= \mathsf{PBKDF}(\mathit{initializationCode}, \text{"}\texttt{authid}\text{"} + \texttt{eeId}), \\ \mathit{symmetricKey} &= \mathsf{PBKDF}(\mathit{initializationCode}, \text{"}\texttt{authpassword}\text{"} + \texttt{eeId}). \end{aligned}$$

- The voting client sends *authenticationId* to the web server, which responds with an encrypted start voting key $\texttt{SVK}_{\texttt{id}}$ and the keystore $\texttt{VCks}_{\texttt{id}}$.
- The voting client uses *symmetricKey* to decrypt the encrypted start voting key.
- The voting client opens $\texttt{VCks}_{\texttt{id}}$ using $\texttt{SVK}_{\texttt{id}}$.

In the light of this direct comparison, we have several important remarks. First, it is obvious that the implemented authentication method is completely underspecified. It includes several cryptographic elements (authentication key, initialization code, authentication ID), which are not even mentioned in the specification. The length of the authentication key, for example, is a very critical security parameter, which cannot be left unspecified. What is also completely unclear is the selection of the key generator class, and, more generally, the question why there are two such classes.

Second, it is difficult to understand the benefits of this complicated implementation. Given the specification, the only open question is the selection of the right keystore by the voting server. In principle, since the keystores are encrypted, the voting server could send all keystores to the voting client, and the selection is performed locally in a trial-and-error procedure. However, for efficiency reasons, it would be better for the voting client to simply send an identifier for the keystore to the voting server, for example the hash value of the start voting key, another (shorter) unique id such as the index of the

keystore in the list of keystores, or the verification card ID $\texttt{vc}_{\texttt{id}}$. Clearly, the currently implemented solution is unnecessarily complicated for no obvious benefit.

In the following list, we summarize some additional remarks about the implementation of the authentication process:

- The specification gives the impression that all key derivations are computed using Argon2. But in the implemented procedure, PBKDF2 is used (see authenticate.js).
- The PBKDF2 parameters do not depend on the security level.
- For computing *authenticationId* and *symmetricKey*, the same PBKDF2 salt is used for all voters of a given election.
- The implementation of the authentication procedure is even more complicated than described above. It includes additional confusing steps with tokens, client/server challenge messages, signatures, and certificates. Given the time constraint for our assessments, we were unable to conduct a full analysis of these steps.

To conclude the discussion of this topic, we recommend to entirely remove the current implementation and to work out a new solution from scratch based on a clear and well documented concept.

B.3.4. Hybrid Elections with Conventional Voting Channels

In the discussion added to Section 5.2.4, an entirely new aspect has been introduced. The problem comes from the fact that multiple voting channels are offered to the voters in Switzerland simultaneously. This creates a synchronization problem across multiple voting channels, which must be solved carefully such that voters cannot submit votes over multiple channels. In [HDKL18, Section 3], this problem has been analyzed and possible extensions of the election and verification processes have been proposed. The discussed subtleties in [HDKL18] demonstrate the difficulty of this problem.

According to the new explanations given in Section 5.2.4, the so-called *voting card status* is managed by the untrusted voting server. As soon as a voter submits and confirms a ballot electronically, or whenever a paper ballot is registered at the cantonal or communal election office, the voting card status is updated. Unfortunately, the possible values for the voting card status are not further specified, but we assume it will be one of the following four states: *not-voted* (initial state), *voted-electronically* (terminal state), *voted-by-mail* (terminal state), and *voted-in-person* (terminal state). The idea then is to block the submission of a ballot, if the status is already in a terminal state.

Under the premise that our understanding of this topic is correct, we see a number of critical problems. The first comes from the responsibility given to the untrusted voting server to manage the voting card status. This implies that by changing the status maliciously, the voting server can block voters arbitrarily in all voting channels. The problem is that such malicious changes cannot be detected, or more generally, that the

correctness of the voting card status cannot be publicly verified. Furthermore, if the cantonal or communal election offices can invoke a status change at the voting server, then a strong authentication mechanism is required to prevent an attacker from invoking such changes maliciously. However, such an authentication mechanism is not specified. Finally, we are concerned that synchronization problems may occur if a large number of ballots are cast at more or less the same time, which could then lead to unintended duplicate votes. We cannot describe such a scenario precisely, but they are typical for asynchronous processes.

We assume that a discussion of this topic has been added to Section 5.2.4 mainly to comply with the following requirements from [OEV, No. 4.11] and [OEV, No. 11.6], respectively (both requirements and corresponding notes from [ExpRep] are quoted in Section 5.2.4):

> *"As long as the system has not registered confirmation of a definitive electronic vote, the voter may still choose to cast his or her vote via a conventional voting channel."*

> *"The system allows the polling card to be used to determine whether someone has cast an electronic vote."*

However, from these statements we do not conclude that this functionality should be implemented by the untrusted voting server, even if a note in [ExpRep] says that it is not necessary to specify this functionality under the same trust assumptions as for complete verifiability.

B.3.5. Proof of Non-Participation and Handling Inconsistent Views

After an election, abstaining voters must have the possibility to check no vote has been cast on their behalf. In [HKLD22b, Section 2.3], we have already discussed this topic. We suggested a solid solution based on *vote abstention codes*, but our recommendation has not been implemented. The current solution described in the new Section 6.2.6 is simpler, but it requires stronger trust assumptions. The idea is that a trustworthy cantonal office knows the list of voting card IDs of all submitted and confirmed votes. While the new Algorithm 6.8 (RequestProofNonParticipation) is trivial, it is unclear how this service will work in practice. For example, given the definition of `VCard` in Figure 6, it seems as if the voting card ID $\texttt{vc}_{\texttt{id}}$ is not directly known to the voter, so it remains unclear how the cantonal office receives the right value. The given information in the new Section 6.2.6 is therefore incomplete.

Another entirely new aspect is discussed in the new Section 6.2.7. The addressed problem is a situation, in which not all control components have the same view after the election phase. This could happen for various reasons, intentionally or unintentionally. In such

a case, the check performed by algorithm VerifyMixDecOnline will fail for at least one control component and the tally procedure is aborted. To recover from such a situation, an additional trustworthy protocol party called *dispute resolver* is introduced. The idea to check the inclusion of each submitted and confirmed vote by a third party, who calls two new algorithm SendVoteAgreement and ConfirmVoteAgreement for that purpose. The goal is to come up with an agreed list of confirmed votes, which is accepted by all control components as input for re-running the tally phase. While we agree that dispute resolving procedures can help to increase the robustness of a cryptographic protocol, we are not sure if it is necessary to include it as part of the protocol description, and also if the OEV regulations allow the introduction of additional trustworthy parties. Note that there may be other disputes, for example during the setup phase. Therefore, we recommend moving this discussion into a separate section.

B.3.6. Randomness Generation

As announced in the project's main CHANGELOG.md file of the December release, we found a new configuration sub-folder security in the root directory of the E-voting component. It contains two sub-folders `security/windows` and `security/linux` with corresponding `java.security` properties files. This is a response to our comments in [HKLD22c, Section 2.7] about the importance of a reliable and high-quality entropy source. There, we made several recommendations for improving the reliability and quality of the entropy source, for example by introducing health tests or CPU time jitter entropy. So far, our recommendations have not been considered.

Copying the Java `java.security` properties files into the project's main directory has no security impact, because it does not prevent Java from reading the security properties from the original file in the JRE installation directory. In the README.md file of the security sub-folder, Swiss Post justifies this step as a measure for increasing the system's auditability:

> *"The Swiss Post Voting System relies on the operating system to select a high-quality PRNG when performing cryptographic operations. For increased auditability and to ensure that an appropriate PRNG is selected in practice, this folder contains the Java security configuration for the Linux and Windows operating systems used in the deployed system."*

With respect to the selected entropy source, we can only observe that the copied files contain correct standard entries `"securerandom.source=file:/dev/random"`, but this says nothing about the configuration of the machine on which the deployed system is running. To claim that these files are helpful for increasing the system's auditability is therefore misleading. Note that even if the implementation of `SecureRandom` is properly configured to take the entropy from `/dev/random`, it is still possible to redirect the entropy

source to something else on the operating system layer. We discussed such possibilities in our report from last year.

We conclude this discussion by emphasizing one more time the importance of this subtle topic. We believe that the randomness generation should never be delegated to a single not fully reliable entropy source, and that a safety net for detecting the most obvious failure cases should always be installed on top of it. We have expressed our concerns already in [HKLD22c, Section 2.7], we repeated them in Subsection 2.2.7, and we still have them after looking at the December release and finishing the second assessment round.

B.4. Verifier

The verifier specification [VerSpec] has been substantially extended in the November and December releases. For example, several adjustments were necessary to incorporate the write-ins. Furthermore, as a response to our remark in Subsection 3.4 that some verification steps were not sufficiently well specified, many more details are now provided. A total of 41 new pseudocode verification algorithms have been introduced for that purpose. The following list gives an overview of all the major changes that we observed in the specification document:

- New Section 2.4 on *Manual Checks by the Auditors* added.
- New input file `setup/setupComponentPublicKeys.json` added to Table 2.
- Message *SetupComponentConfig* (signed by setup component) in Table 3 renamed into *CantonConfig* (signed by canton).
- Content of message *SetupComponentPublicKeys* in Table 3 extended with missing zero-knowledge proofs.
- New message *ElectionEventContext* added to Table 3.
- New input file `tally/eCH-0222.xml` added to Table 4 and a new authenticity check *TallyComponentEch0222* added to Table 5.
- Content of the message *TallyComponentVotes* in Table 5 extended from $L_{\texttt{votes}}$ to $(L_{\texttt{votes}}, L_{\texttt{decodedVotes}}, L_{\texttt{writeIns}})$. Lines 5–6 of Verification 2.11, Lines 2–5 of Algorithm 4.2, and Lines 8–9 of Algorithm 4.3 adjusted accordingly to include checking the write-ins.
- New explicit Verifications 2.01 to 2.08 added to Section 3.2 (authenticity checks of setup verification).
- New explicit Verifications 3.01 to 3.15 added to Section 3.3 (consistency checks of setup verification).

- New explicit Verifications 7.01 to 7.07 added to Section 4.2 (authenticity checks of tally verification).
- New explicit Verifications 8.01 to 8.11 added to Section 4.3 (authenticity checks of tally verification).
- Setup component public keys added to the inputs of Verification 10.01 (Line 3 extended accordingly), Algorithm 4.1, and Verification 10.02 (Line 2 extended accordingly).

In an additional remark in Section 2.1, a justification is given for the absence of pseudocode algorithms for the two top-level verification procedures VerifyConfigPhase and VerifyTally. This is another response to one of our remarks in Subsection 3.4.

While reviewing the source code of the verifier component, we located the added verification steps and their implementations. However, due to the limited timeframe of the second assessment round and the dynamic nature of the verifier specification, we were unable to conduct an equally thorough audit of these steps, compared to the algorithms of the crypto-primitives and e-voting components. As in previous versions, a catalog of implemented verification steps is given in the README.md file of the verifier-backend project. Compared to the October release, the number of entries in this catalog has been increased from 39 (24 for verifying the setup and 15 for the tally) to 51 (30 for verifying the setup and 21 for the tally). The extended catalog of the latest version is shown in Figure 12.

As each entry in the catalog has a name and an assigned identifier, we would have expected that theses names and identifiers match with the pseudocode verifications in the specification. For the names, in most cases, this is actually the case now, but not for the identifiers. For example, the entry VerifySignatureCantonConfig with identifier 201 clearly correspond to Verification 2.02, so the identifier is not aligned with the numbering (for no obvious reason, the numbers of most 2.0X, 3.0X, and 5.0X verifications are shifted by 1 compared to corresponding catalog identifiers). In some cases, the misalignment is even more visible, for example in the case of VerifyEncryptedPCCExponentiationProofs with identifier 504 and number 5.21 or in the case of VerifySignatureControlComponentBallotBox with identifier 200 and number 7.01. Note that neither the identifiers nor the names in the catalog are unique, which is very confusing. We already criticized the lack of a clear concept, so we are surprised to see that this still seems to be the case. Even some fundamental terminology is not used consistently, for example the redundant terms *config* and *setup* are mixed up on many occasions. We recommend to scan the whole document for such inconsistencies and to remove them in a systematic manner.

In the new Section 2.4 about the manual checks of the auditors, it is not always clear, what exactly the auditor has to accomplish. The following quote gives an example of a statement that is contradictory in itself:

Phase	Category	Id	Name of the verification
Setup	Completeness	100	VerifySetupCompleteness
Setup	Authenticity	200	VerifySignatureSetupComponentEncryptionParameters
Setup	Authenticity	201	VerifySignatureCantonConfig
Setup	Authenticity	202	VerifySignatureSetupComponentPublicKeys
Setup	Authenticity	203	VerifySignatureControlComponentPublicKeys
Setup	Authenticity	204	VerifySignatureSetupComponentVerificationData
Setup	Authenticity	205	VerifySignatureControlComponentCodeShares
Setup	Authenticity	206	VerifySignatureSetupComponentTallyData
Setup	Authenticity	207	VerifySignatureElectionEventContext
Setup	Consistency	300	VerifyEncryptionGroupConsistency
Setup	Consistency	301	VerifySetupFileNamesConsistency
Setup	Consistency	302	VerifyCCrChoiceReturnCodesPublicKeyConsistency
Setup	Consistency	303	VerifyCCmElectionPublicKeyConsistency
Setup	Consistency	304	VerifyCcmAndCcrSchnorrProofsConsistency
Setup	Consistency	305	VerifyChoiceReturnCodesPublicKeyConsistency
Setup	Consistency	306	VerifyElectionPublicKeyConsistency
Setup	Consistency	307	VerifyPrimesMappingTableConsistency
Setup	Consistency	308	VerifyElectionEventIdConsistency
Setup	Consistency	309	VerifyVerificationCardSetIdsConsistency
Setup	Consistency	310	VerifyFileNameVerificationCardSetIdsConsistency
Setup	Consistency	311	VerifyVerificationCardIdsConsistency
Setup	Consistency	312	VerifyTotalVotersConsistency
Setup	Consistency	313	VerifyNodeIdsConsistency
Setup	Consistency	314	VerifyChunkConsistency
Setup	Evidence	500	VerifyEncryptionParameters
Setup	Evidence	501	VerifySmallPrimeGroupMembers
Setup	Evidence	502	VerifyVotingOptions
Setup	Evidence	503	VerifyKeyGenerationSchnorrProofs
Setup	Evidence	504	VerifyEncryptedPCCExponentiationProofs
Setup	Evidence	505	VerifyEncryptedCKExponentiationProofs
Tally	Completeness	100	VerifyTallyCompleteness
Tally	Authenticity	200	VerifySignatureControlComponentBallotBox
Tally	Authenticity	201	VerifySignatureControlComponentShuffle
Tally	Authenticity	202	VerifySignatureTallyComponentShuffle
Tally	Authenticity	203	VerifySignatureTallyComponentVotes
Tally	Authenticity	204	VerifySignatureTallyComponentDecrypt
Tally	Authenticity	205	VerifySignatureTallyComponentEch0110
Tally	Authenticity	206	VerifySignatureTallyComponentEch0222
Tally	Consistency	300	VerifyConfirmedEncryptedVotesConsistency
Tally	Consistency	301	VerifyCiphertextsConsistency
Tally	Consistency	302	VerifyPlaintextsConsistency
Tally	Consistency	303	VerifyVerificationCardIdsConsistency
Tally	Consistency	304	VerifyBallotBoxIdsConsistency
Tally	Consistency	305	VerifyFileNameBallotBoxIdsConsistency
Tally	Consistency	306	VerifyNumberConfirmedEncryptedVotesConsistency
Tally	Consistency	307	VerifyElectionEventIdConsistency
Tally	Consistency	308	VerifyNodeIdsConsistency
Tally	Consistency	309	VerifyFileNameNodeIdsConsistency
Tally	Consistency	310	VerifyEncryptionGroupConsistency
Tally	Evidence	500	VerifyOnlineControlComponents
Tally	Evidence	501	VerifyTallyControlComponent

Figure 12: Catalog of verification tests as defined in the README.md file of the verifier backend.

> *"Moreover, the auditors can assume that the configuration of the election event—signed by the canton [...]—is correct. Nevertheless, the auditors should also manually check that the information in the election event configuration is correct."*

We conclude from our analysis of the verifier component that many parts are still under construction and refinement. We can confirm that changes made to the cryptographic protocol have been updated and that some of the underspecified verification steps have been clarified, but for example most of the statements in textual form listed in Appendix A.4.1 are still present in the current version. We can also confirm that the most obvious and relevant checks are all present, but even after having examined all specified algorithms and their implementations, we cannot confirm that the implemented catalog of verification steps covers really everything. The specification document does not yet provide stringent evidence for the verification process as a whole, in the sense that it does not prove the existence of a complete and unbroken verification chain. We are also not convinced that the information given in [VerSpec] is sufficient for a third party to develop an independent verifier.

B.5. Recapitulation

Our assessment of the November and December releases was conducted under enormous time pressure. For analyzing and evaluating the December release, for example, we only had a period of less than 20 working days and an overlap with both the end of the year and the end of the academic semester. In our findings reported in this second addendum, we demonstrate that many important parts of the system have been updated in both the specification and the code, but given the tight schedule, we were not always able to conduct our analysis as carefully as we would have wished. We are also not fully convinced, that evaluating a system of such great complexity in the proposed manner, i.e., with new releases and additional assessments rounds every couple months, is an adequate process for checking its security properties. The greatest challenge that we faced in this process was keeping a focus on the big picture while looking at all the tiny technical details. In some cases, we are unable to do both at the same time.

The most general conclusion from this second additional evaluation round comes from our impression that almost all system components are still work in progress. We discovered numerous last-minute code changes and enhancements to the documentation, but we also observed that some of these changes were not implemented with sufficient care and dedication. A good example is the information added to Section 5.1 about voter authentication, which seems to contribute to the clarification of this aspect. However, our detailed code analysis showed quite the opposite, namely that the implemented procedure is completely underspecified. Such discrepancies between specification and code are clearly not in in the sense of [OEV, Art. 25.2.8], which requires that "*the cryptographic protocol, specification, design and source code are aligned*". Therefore, despite the progress and improvements that we also observed in many places, it does not seem that the system has already reached the expected level of maturity.

References

[AB17] J. Alwen and J. Blocki. Towards practical attacks on Argon2i and balloon hashing. In A. Sabelfeld and M . Smith, editors, *2nd IEEE European Symposium on Security and Privacy*, pages 142–157, Paris, France, 2017.

[BCG+15] D. Bernhard, V. Cortier, D. Galindo, O. Pereira, and B. Warinschi. A comprehensive analysis of game-based ballot privacy definitions. In L. Bauer and V. Shmatikov, editors, *SP'15, 36th IEEE Symposium on Security and Privacy*, pages 499–516, San Jose, USA, 2015.

[Blo18] J. Bloch. *Effective Java.* Addison-Wesley, 3rd edition, 2018.

[HDKL18] R. Haenni, E. Dubuis, R. E. Koenig, and P. Locher. Process models for universally verifiable elections. In R. Krimmer, M. Volkamer, V. Cortier, R. Goré, M. Hapsara, U. Serdült, and D. Duenas-Cid, editors, *E-Vote-ID'18, 3rd International Joint Conference on Electronic Voting*, LNCS 11143, pages 84–99, Bregenz, Austria, 2018.

[HKLD22a] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote protocol specification – version 3.3. *IACR Cryptology ePrint Archive*, 2017/325, 2022.

[HKLD22b] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. Examination of the Swiss Post Internet Voting System – Scope 1: Cryptographic Protocol. Technical report, Bern University of Applied Sciences, 2022.

[HKLD22c] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. Examination of the Swiss Post Internet Voting System – Scope 2: Software. Technical report, Bern University of Applied Sciences, 2022.

[Wie03] M. J. Wiener. Safe prime generation with a combined sieve. *IACR Cryptology ePrint Archive*, 2003/186, 2003.